Wissenswertes zu Bits, Bytes und Flags

Mit dieser Abhandlung werden Sie in die Welt der Bits und Bytes eingeführt. Anfangen wollen wir mit ein paar Grundlagen, auch wenn Ihnen diese vermutlich längst bekannt sind.

Die Welt des Programmieren beginnt mit Bits und Bytes. **Bit**, das ist ein Kunstwort. Es steht für die Abstraktion zweier elektrischer Signalzustände: so kann ein elektrisches Gerät ausgeschaltet oder eingeschaltet sein.

Und genau solche Signalzustände werden durch Bits beschrieben.

- Für den Betriebszustand *»**ausgeschaltet**«* wird einem Bit der Zustandswert (auch Statuswert genannt) *»Null«* ***0*** zugeschrieben. (Alternativ wird dieser auch schon mal mit *»aus«*, *»off«*, *»niedrig«*, *»low«* *L*, *»falsch«* *F* oder *»false«* *F* bezeichnet.)
- Für den Betriebszustand *»**eingeschaltet**«* wird einem Bit der Zustandswert *»Eins«* ***1*** zugeschrieben. (Alternativ wird dieser auch schon mal mit *»an«*, *»on«*, *»hoch«*, *»high«* *H*, *»wahr«* *W* oder *»true«* *T* bezeichnet.)

0	ein Bit mit dem Statuswert »*0*« (auch »*aus*«, »*of*«, »*falsch*« oder »*false*«)
1	ein Bit mit dem Statuswert »*1*« (auch »*an*«, »*on*«, »*wahr*« oder »*true*«)

Um die Verarbeitung von Bits mittels Computer zu vereinfachen, werden diese zu übersichtlichen Gruppen arrangiert. Solche Gruppen werden **Byte** genannt. Byte, das ist wiederum ein Kunstwort. Die Zahl der Bits, die zu einem Byte zusammengefaßt werden, kann variieren und ist durch die Hardwarekonzeptionen eines jeweiligen Computers bestimmt. So kennt man Entwicklungen, die beispielsweise auf Festlegung von

- einem Byte zu 4 Bit,
- einem Byte zu 8 Bit oder
- einem Byte zu 9 Bit

basieren.

EVO PDF Tools Demo

3	2	1	0	
1	1	0	0	ein **4-Bit-Byte**

7	6	5	4	3	2	1	0	
1	0	0	1	1	1	0	0	ein **8-Bit-Byte**

8	7	6	5	4	3	2	1	0	
1	1	0	0	1	1	1	0	0	ein **9-Bit-Byte**

Natürlich möchten wir Sie auch auf Beispiele solcher technischen Festlegungen hinweisen:

- mit Bytes von je 4 Bit:
  - von Intel das Modell 4004;
  - von NEC die Prozessorfamilie µPD75X;
  - von Texas Instruments die Serie TMS1000;
  - von Hewlett-Packard die HP-Saturn-Familie.

- mit Bytes von 9 je Bit:
  - die Minicomputer PDP-1, PDP-4, PDP-7, PDP-9 und PDP-15 (basierend auf 2 Byte zu jeweils 9 Bit);
  - die PDP-10 (basierend auf 2 Byte und 4 Byte zu jeweils 9 Bit);
  - diverse UNIVAC-Computer, beispielsweise die UNIVAC 418;
  - das IBM 7700 Datenerfassungssystem;
  - das BCL Molecular 18;
  - der NASA-Standard-Raumflugcomputer NSSC-1.

- mit Bytes von 8 je Bit:
  - Dies ist die geläufigste Bytebreite. Auf ihr basieren die meisten Home- und Minicomputer, als auch die IBM-kompatiblen Personal Computer Systeme.

Was können Computer nun mit diesen Bits machen?

Computer können grundlegende Operationen auf Bits anwenden, von denen wir Ihnen hier die wichtigsten vorstellen. Diese Operationen führen sowohl deutsche als auch englische Namensbezeichnungen. Da unsere Betrachtungen jedoch aufs Programmieren bzw. auf Programmiersprachen abzielen, wenden wir uns der englischen Sprache entlehnten Namensbezeichnungen zu. Diese werden Sie in adäquater Form im Sprachumfang der meisten Programmiersprachen wiederfinden.

Hinsichtlich der Bit-Operationen werden wir Ihnen diese per Text und Wertetabelle vorstellen. Der Text beschreibt die Bit-Operation, und die Wertetabelle stellt Eingangs- und Ausgangsgröße (also *Eingangs-Bit* und *Ausgangs-Bit*) einander gegenüber. Die Ausgangsgröße (das Ausgangs-Bit) ist zugleich das Ergebnis der Bit-Operation. Der Vergleich von Eingangs- und Ausgangs-Bit zeigt die im Text beschriebene Wirkungsweise der Bit-Operation. Zudem ist jede Zeile einer Wertetabelle farbig hinterlegt:

- Eine Zeile ist weiß hinterlegt, wenn das Ausgangs-Bit (das Ergebnis) dieser Zeile den Bit-Status 0 ergibt.
- Eine Zeile ist grau hinterlegt, wenn das Ausgangs-Bit (das Ergebnis) dieser Zeile den Bit-Status 1 ergibt.

Diese Farbgebung hat keinen technischen Hintergrund, sondern soll alleine Ihrer Orientierung dienen. Zudem kann die Farbgebung hilfreich sein, wenn Sie verschiedene Wertetabellen rasch miteinander vergleichen möchten.

Unterhalb der Wertetabelle stellen wir Ihnen Formelschreibweisen für die genannte Bit-Operation vor. Bereits beim ersten Hinsehen werden Ihnen Unterschiede in diesen Schreibweisen auffallen. Trotz dieser Unterschiede in den Schreibweisen, drücken sie jeweils die gleiche Bit-Operation aus. Zu jeder Wertetabelle geben wir fallweise drei oder vier unterschiedliche Formelschreibweisen an, und bedienen uns dabei der folgenden Schreibformen:

Schreibweise	Bedeutung
algebraisch	Eine mathematische Schreibweise der *Algebra* mit dem entsprechenden mathematischen Operator-Symbol(en). Beispiel: $E := A \wedge B$ EVO PDF Tools Demo Wo ein Teilausdruck vorrangig zu behandeln ist, wird dieser in Klammern gesetzt. Beispiel: $E := A \wedge (B \vee C)$
infix	Eine Schreibweise vergleichbar infix-basierter Programmiersprachen. Die *Infix-Notation* ist eine Schreibweise, bei der ein Operator zwischen seinen Operanden notiert wird. Beispiel: $E := A \text{ and } B$ Wo ein Teilausdruck vorrangig zu behandeln ist, wird dieser in Klammern gesetzt. Beispiel: $E := A \; and \; (B \; or \; C)$
postfix	Eine Schreibweise vergleichbar postfix-basierter Programmiersprachen. Die *Postfix-Notation* ist eine Schreibweise, bei der ein Operator nach (d.h. hinter) seinen Operanden notiert wird. Beispiel: $E := A \; B \text{ and}$ Diese Schreibweise kennt keine Klammersetzung. Ihre Ausdrücke werden konsequent von links nach rechts (und dann ggf. noch von oben nach unten) abgearbeitet. Beispiel: $E := B \; C \; or \; A \; and$
	Eine Schreibweise vergleichbar funktional-basierter Programmiersprachen. Die *funktionale Notation* ist eine Schreibweise, bei der Funktionen an Stelle von Operatoren verwendet werden. Dabei werden die Operanden diesen Funktionen in festgelegter Reihenfolge als Parameter übergeben. Üblich festgelegt ist, daß zunächst der Funktionsname notiert wird, gefolgt von einer öffnenden Klammer. Dann folgt die Nennung der durch Komma separierten Parameter in festgelegter Reihenfolge; und eine schließende Klammer schließt die

Schreibweise	Bedeutung
funktional	Konstruktion ab. Beispiel: $E := \text{and}(A, B)$ Selbstverständlich dürfen auch Funktionen als Parameter übergeben werden, jedoch sind diese zuvor auszuwerten, so das letztlich deren Ergebnis als Parameter übergeben wird. Auf diese Weise lassen sich Funktionen ineinander verschachteln, und solche Verschachtelungen sind von innen nach außen hin aufzulösen. Beispiel: $E := \text{and}(A, \text{or}(B, C))$ Funktionale Schreibweisen finden sich in funktionalen Programmiersprachen, teilweise aber auch in den Zellenformeln von Tabellenkalkulationen wie »LibreOffice Calc« oder »Microsoft Excel«.

Grundlegende Bit-Operationen

UND-Verknüpfung

Bei der *UND-Verknüpfung* ist das Ergebnis-Bit *E* ist nur dann 1, wenn die Statuswerte der Eingangs-Bits *A* und *B* beide 1 sind; andernfalls ist das Ergebnis-Bit 0.

Legen wir dazu eine Wertetabelle an, ergibt sich folgendes Bild:

AND		
A	**B**	**E**
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

EVO PDF Tools Demo

E := (A∧B)	(-algebraisch-)
E := (A *and* B)	(-infix-)
E := A B *and*	(-postfix-)
E := *And*(A , B)	(-funktional-)

Ein alternative Bezeichnungen der UND-Verknüpfung ist:

- *AND-Verknüpfung*

ODER-Verknüpfung

Bei der *ODER-Verknüpfung* ist das Ergebnis-Bit *E* ist genau dann 1, wenn wenigstens eines der Eingangs-Bits *A* und *B* den Statuswert 1 hat; andernfalls ist das Ergebnis-Bit 0.

Ihre Wertetabelle lautet:

OR		
A	**B**	**E**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

E := (A∨B)	(-algebraisch-)
E := (A *or* B)	(-infix-)
E := A B *or*	(-postfix-)
E := *Or*(A , B)	(-funktional-)

Eine alternative Bezeichnungen der ODER-Verknüpfung ist:

- *OR-Verknüpfung*

Exklusiv-ODER- Verknüpfung

Bei der *Exklusiv-ODER-Verknüpfung* ist das Ergebnis-Bit *E* ist genau dann 1, wenn lediglich genau eines der der Eingangs-Bits *A* und *B* den Statuswert 1 hat; andernfalls ist das Ergebnis-Bit 0.

Die Wertetabelle lautet:

XOR		
A	**B**	**E**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

$E := (A \veebar B)$	(-algebraisch-)
$E := (A \ xor \ B)$	(-infix-)
$\mathbf{E := A\ B\ xor}$	(-postfix-)
$E := Xor(A, B)$	(-funktional-)

Alternative Bezeichnungen der Exklusiv-ODER-Verknüpfung sind:

- XODER-Verknüpfung
- Exklusiv-OR-Verknüpfung
- *XOR-Verknüpfung*

Bit-Operationen mit Invertierung

Invertieren eines Bit

EVO PDF Tools Demo

Beim *Invertieren* nimmt das Ergebnis-Bit *E* jeweils den umgekehrten Statuswert gegenüber dem Statuswert des Eingangs-Bit *A* an. Das Ergebnis-Bit *E*

- ist 0, wenn *A* den Statuswert 1 hat; und
- ist 1, wenn *A* den Statuswert 0 hat.

Damit ergibt sich folgende Wertetabelle:

INVERT	
A	**E**
0	**1**
1	**0**

$E := (\neg A)$	(-algebraisch-)
$E := (invert \ A)$	(-infix-)
$\mathbf{E := A\ invert}$	(-postfix-)
$E := Invert(A)$	(-funktional-)

Häufig genutzt Alternativbezeichnungen für das Invertieren eines Bits sind:

- *Negieren* (Negierung)
- Umkehren (Umkehrung)
- Vertauschen (Vertauschung)
- *NICHT-Operation*
- *NOT-Operation*
- *NEGATE-Operation*

Unglücklicherweise überschneiden sich diese alternativen Bezeichnungen häufig mit namensgleichen oder zumindest namensähnlichen Bezeichnungen zu Operationen auf Zahlen und Wahrheitswerten der Booleschen Algebra (welche in vielen Programmiersprachen durch Zahlenwerte repräsentiert wird). Und diese Operationen sind aufwendiger, als das bloße Invertieren von Bits. Um diesbezüglich Eindeutigkeit zu erreichen, legen wir folgende

Begriffe fest:

- Mit ***Invertieren*** bzw. ***INVERT*** bezeichnen wir ausschließlich eine Operation zum Umkehren von Bits.
- Mit ***Negieren*** bzw. ***NEGATE*** bezeichnen wir ausschließlich eine Operation für den Vorzeichenwechsel bei Zahlen.

Beispiel: Wenn wir die positive Zahl 37 negieren, wird daraus die negative Zahl -37. Und wenn wir die negative Zahl -37 nochmals negieren, wird daraus wieder die positive Zahl 37.

- Mit ***NICHT*** bzw. ***NOT*** bezeichnen wir ausschließlich eine Operation zum Umkehren von Wahrheitswerten der Booleschen Algebra (den sogenannten »Flags«). Der Kontext dieser Operation bezieht sich ausdrücklich auf die Boolesche Algebra, und zwar selbst dann, wenn dafür Bits invertiert werden sollten.

Passend zu diesen Festlegungen haben wir die hier besprochene Bit-Operation mit der Überschrift **»Invertieren eines Bit«** versehen und haben jede verwirrende Wortwahl, die unserer Festlegung entgegenstände, in der Überschrift vermieden.

Die Wirkungsweise von INVERT braucht nicht in einem Computer oder einer Programmiersprache implementiert sein, den sie läßt sich problemlos mittels eines auf 1 gesetzten Bits und der XOR-Operation durchführen und liefert dabei immer das korrekte Ergebnis:

INVERT	
A	E
0	**1**
1	**0**

$E := (A \veebar 1)$	(-algebraisch-)
$E := (A \ xor \ 1)$	(-infix-)
$\mathbf{E := A\ 1\ xor}$	(-postfix-)
$E := Xor(A, 1)$	(-funktional-)

Inhibition

EVO PDF Tools Demo

Das Wort »inhibitieren« bedeutet soviel wie »hemmen« oder auch »zurücksetzen«. Damit beschreibt die Inhibition eine Bit-Operation, die ein Bit hindern (hemmen) kann, in ein Ergebnis einzufließen.

Die *Inhibition* nutzt also zwei Eingangs-Bit (*A* und *B*). Das zweite Eingangs-Bit *B* entscheidet dabei, ob der Zustandswert des Eingangs-Bit *A* in das Ergebnis-Bit *E* einfließt oder nicht. Hat das Eingangs-Bit *B* den Wert 0, so wird der Zustandswert des Eingangs-Bit *A* in das Ergebnis-Bit *E* übernommen. Hat das Eingangs-Bit *B* hingegen den Wert 1, so unterdrückt (hemmt) die Inhibition die Übernahme des Zustandswertes von Eingangs-Bit *A* und reicht statt dessen den Zustandswert 0 an das Ergebnis-Bit *E* weiter.

Die diesbezügliche Wertetabelle lautet:

INHIBIT		
A	B	E
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

$E := (A \ inhibit \ B)$	(-infix-)
$\mathbf{E := A\ B\ inhibit}$	(-postfix-)
$E := Inhibit(A, B)$	(-funktional-)

Technisch betrachtet ist die Inhibition nichts anderes als eine UND-Verknüpfung des Eingangs-Bit *A* mit dem invertierten Zustandswert von Eingangs-Bit *B* :

INHIBIT		
A	B	E
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

$E := (A \wedge (\neg B))$	(-algebraisch-)

E := (A ∧ (¬ B))		(-algebraisch-)
E := (A *and* (*invert* (B)))		(-infix-)
E := A B *invert and*		(-postfix-)
E := *And*(A , *Invert*(B))		(-funktional-)

Anmerkung: In der Fachliteratur zur Inhibition ist nicht festgelegt, welches der beiden Eingangs-Bit (*A* oder *B*) invertiert wird. Je nach Anwendungsfall entscheidet man sich für eines der beiden und führt anschließend die UND-Verknüpfung mit dem anderen Eingangs-Bit durch. Wir legen uns jedoch darauf fest, daß wir die Inhibition stets mit der Invertierung des zweiten Eingangs-Bit *B* durchführen. Und falls einmal die andere Variante erforderlich sein sollte, brauchen wir nur die Reihenfolge der Eingangs-Bits *A* und *B* gegeneinander vertauschen.

NAND-Verknüpfung

Die *NAND-Verknüpfung* beruht auf einer AND-Verknüpfung, deren Ausgangs-Bit invertiert wird. Bei dieser Verknüpfungsfolge erhält das Ergebnis-Bit *E* den Statuswert 1, solange die Eingangs-Bits *A* und *B* nicht gemeinsam jeweils den Statuswert 1 haben. In diesem Fall wäre das Ergebnis-Bit 0.

Die Wertetabelle lautet:

NAND		
A	**B**	**E**
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

E := (A *nand* B)		(-infix-)
E := A B *nand*		(-postfix-)
E := *Nand*(A , B)		(-funktional-)

Gebräuchliche Alternativbezeichnungen der der NAND-Verknüpfung sind:

EVO PDF Tools Demo

- Invertierte-UND-Verknüpfung
- Invertierte-AND-Verknüpfung
- Negierte-UND-Verknüpfung
- Negierte-AND-Verknüpfung
- NICHT-UND-Verknüpfung
- NICHT-AND-Verknüpfung
- NOT-UND-Verknüpfung
- *NOT-AND-Verknüpfung*

Die Wirkungsweise von NAND braucht nicht in einem Computer oder einer Programmiersprache implementiert sein, den sie läßt sich mittels einer Kombination aus AND und INVERT durchführen:

NAND		
A	**B**	**E**
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

E := (¬ (A ∧ B))		(-algebraisch-)
E := (*invert* (A *and* B))		(-infix-)
E := A B *and invert*		(-postfix-)
E := *Invert*(*And*(A , B))		(-funktional-)

NOR-Verknüpfung

Die *NOR-Verknüpfung* beruht auf einer OR-Verknüpfung, deren Ausgangs-Bit invertiert wird. Bei dieser Verknüpfungsfolge ist das Ergebnis-Bit *E* dann 1, wenn die Eingangs-Bits *A* und *B* gemeinsam jeweils den Statuswert 0 haben; andernfalls ist das Ergebnis-Bit 0.

Ihre Wertetabelle lautet:

NOR		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

E := (A *nor* B)		(-infix-)
E := A B *nor*		(-postfix-)
E := *Nor*(A , B)		(-funktional-)

Gebräuchliche Alternativbezeichnungen der der NOR-Verknüpfung sind:

- Invertierte-ODER-Verknüpfung
- Invertierte-OR-Verknüpfung
- Negierte-ODER-Verknüpfung
- Negierte-OR-Verknüpfung
- NICHT-ODER-Verknüpfung
- NICHT-OR-Verknüpfung
- NOT-ODER-Verknüpfung
- *NOT-OR-Verknüpfung*

Die Wirkungsweise von NOR braucht nicht in einem Computer oder einer Programmiersprache implementiert sein, den sie läßt sich mittels einer Kombination aus OR und INVERT durchführen:

NOR		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

EVO PDF Tools Demo

E := $(\neg(A \vee B))$		(-algebraisch-)
E := (*invert* (A *or* B))		(-infix-)
E := A B *or invert*		(-postfix-)
E := *Invert*(*Or*(A , B))		(-funktional-)

XNOR-Verknüpfung

Bei der *XNOR-Verknüpfung* ist das Ergebnis-Bit *E* genau dann 1, wenn die Eingangs-Bits *A* und *B* jeweils einen gleichen Statuswert haben; andernfalls ist das Ergebnis-Bit 0.

Die Wertetabelle lautet:

XNOR		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

E := (A *xnor* B)		(-infix-)
E := A B *xnor*		(-postfix-)
E := *Xnor*(A , B)		(-funktional-)

Gebräuchliche Alternativbezeichnungen der der NOR-Verknüpfung sind:

- Exklusiv-invertierte-ODER-Verknüpfung
- Exklusiv-invertierte-OR-Verknüpfung
- Exklusiv-negierte-ODER-Verknüpfung
- Exklusiv-negierte-OR-Verknüpfung
- Exklusiv-NICHT-ODER-Verknüpfung

- Exklusiv-NICHT-OR-Verknüpfung
- Exklusiv-NOT-ODER-Verknüpfung
- *Exklusiv-NOT-OR-Verknüpfung*

Die Wirkungsweise von XNOR braucht nicht in einem Computer oder einer Programmiersprache implementiert sein, den sie läßt sich mittels einer Kombination aus XOR und INVERT durchführen:

XNOR		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

E := $(\neg(A \veebar B))$	(-algebraisch-)
E := (*invert* (A *xor* B))	(-infix-)
E := A B *xor invert*	(-postfix-)
E := *Invert*(*Xor*(A , B))	(-funktional-)

Bit-Vergleiche

Bit-Vergleiche vergleichen die Statuswerte zweiter Bits miteinander. Dabei kann der Statuswert des ersten Bit gleich, ungleich, größer als oder kleiner als der Statuswert des zweiten Bit sein.

Ungleich

Das Ergebnis-Bit *E* ist genau dann 1, wenn die Statuswerte der Eingangs-Bits *A* und *B* ungleich sind; andernfalls ist das Ergebnis-Bit 0.

Ihre Wertetabelle lautet:

EVO PDF Tools Demo

<>		
A	B	E
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

E := $(A \neq B)$	(-algebraisch-)
E := (A <> B)	(-infix-)
E := A B <>	(-postfix-)
E := *NotEqual*(A , B)	(-funktional-)

Das Ergebnis dieser Operation stimmt stets mit dem von XOR überein. Somit kann sie durch XOR ersetzt und folgendermaßen durchgeführt werden:

<>		
A	B	E
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

E := (A *xor* B)	(-infix-)
E := A B *xor*	(-postfix-)
E := *Xor*(A , B)	(-funktional-)

Gleich

Das Ergebnis-Bit *E* ist genau dann 1, wenn die Statuswerte der Eingangs-Bits *A* und *B* gleich sind; andernfalls ist das Ergebnis-Bit 0.

Die Wertetabelle lautet:

=		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

E := (A=B)	(-algebraisch-)
E := (A = B)	(-infix-)
E := A B =	(-postfix-)
E := *Equal*(A , B)	(-funktional-)

Das Ergebnis dieser Operation stimmt stets mit dem von XNOR überein. Somit kann sie durch XNOR ersetzt und folgendermaßen durchgeführt werden::

=		
A	B	E
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

E := (A *xnor* B)	(-infix-)
E := A B *xnor*	(-postfix-)
E := *Xnor*(A , B)	(-funktional-)

Größer-als

Das Ergebnis-Bit *E* ist genau dann 1, wenn der Statuswert des Eingangs-Bit *A* größer ist, als der Statuswert des Eingangs-Bit *B* ; andernfalls ist das Ergebnis-Bit 0.

Ihre Wertetabelle lautet:

EVO PDF Tools Demo

>		
A	B	E
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

E := (A>B)	(-algebraisch-)
E := (A > B)	(-infix-)
E := A B >	(-postfix-)
E := *LargerThan*(A , B)	(-funktional-)

Das Ergebnis dieser Operation stimmt stets mit dem von INHIBIT überein und kann daher folgendermaßen ersetzt werden:

>		
A	B	E
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

E := (A *inhibit* B)	(-infix-)
E := A B *inhibit*	(-postfix-)
E := *Inhibit*(A , B)	(-funktional-)

Kleiner-als

Das Ergebnis-Bit *E* ist genau dann 1, wenn die Statuswert des Eingangs-Bit *A* kleiner ist als der Statuswert des Eingangs-Bit *B* ; andernfalls ist das Ergebnis-Bit 0.

Die Wertetabelle lautet:

<		
A	**B**	**E**
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

E :=	*(A<B)*	(-algebraisch-)
E :=	(A < B)	(-infix-)
E := A B <		(-postfix-)
E :=	*LessThan*(A , B)	(-funktional-)

Diese Operation kann durch eine LARGER-THAN-Operation ersetzt werden, bei der man die Reihenfolge der Eingangs-Bit *A* und *B* gegeneinander vertauscht:

<		
A	**B**	**E**
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

E :=	(B > A)	(-infix-)
E := B A >		(-postfix-)
E :=	*LargerThan*(B , A)	(-funktional-)

Das läßt sich natürlich auch mittels der Inhibition verwirklichen:

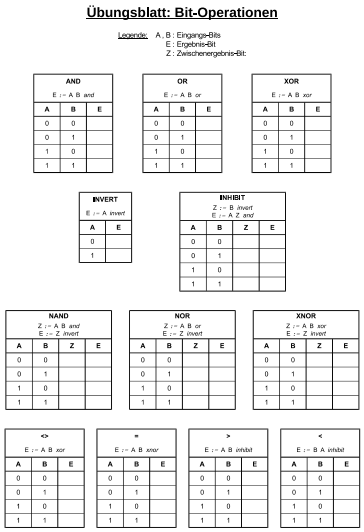
<		
A	**B**	**E**
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

EVO PDF Tools Demo

E :=	(B *inhibit* A)	(-infix-)
E := B A *inhibit*		(-postfix-)
E :=	*Inhibit*(B , A)	(-funktional-)

Übung 1)

Das nachstehende Übungsblatt »Bit-Operationen« enthält einige Wertetabellen. Vervollständigen Sie diese. (Einen Mausklick auf die Grafik des Übungsblattes vergrößert Ihnen dieses.)



Übungsblatt: Bit-Operationen

Legende: A , B : Eingangs-Bits
E : Ergebnis-Bit
Z : Zwischenergebnis-Bit:

AND E := A B *and*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

OR E := A B *or*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

XOR E := A B *xor*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

INVERT E := A *invert*	
A	**E**
0	
1	

INHIBIT Z := B *invert* E := A Z *and*			
A	**B**	**Z**	**E**
0	0		
0	1		
1	0		
1	1		

NAND Z := A B *and* E := Z *invert*			
A	**B**	**Z**	**E**
0	0		
0	1		
1	0		
1	1		

NOR Z := A B *or* E := Z *invert*			
A	**B**	**Z**	**E**
0	0		
0	1		
1	0		
1	1		

XNOR Z := A B *xor* E := Z *invert*			
A	**B**	**Z**	**E**
0	0		
0	1		
1	0		
1	1		

<> E := A B *xor*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

= E := A B *xnor*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

>| E := A B *inhibit*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

<| E := B A *inhibit*		
A	**B**	**E**
0	0	
0	1	
1	0	
1	1	

Operationen auf Byte / Bytegruppen

Nur sehr selten sind Mikroprozessoren und Mikrocontroller so konzipiert, daß sie gezielt einzelne Bits bearbeiten. In der Regel operieren sie auf Gruppen von Bits, den sogenannten Bytes, oder auch auf Gruppen von Bytes. Wie oben ausgeführt, kann die Bytelänge (Bytebreite) - d.h. die Anzahl der Bits für ein Byte - je nach Hardwaredesign variieren. Am gebräuchlichsten ist heutzutage eine Bytelänge zu 8 Bit. Unsere weiteren Betrachtungen beziehen sich daher auf Bytes zu 8 Bit, die aufgrund ihrer 8-Bit-Länge auch **Oktett** genannt werden. Mit **Nibbel** bezeichnet man hingegen Bitfolgen mit einer Länge von 4 Bit. Jede Ziffer einer hexadezimalen Zahl drückt somit ein Nibbel aus.

Wenn die Computerhardware nur auf Bytes bzw. auf Gruppen von Bytes operiert, muß sie so beschaffen sein, daß Bit-Operationen stets auf alle beteiligten Bits eines Bytes bzw. einer Bytegruppe angewendet werden. Doch durch geschicktes Anwenden der auf diesen Bitfolgen arbeitenden Mikroprozessor-Operationen lassen sich dennoch gezielt einzelne Bytes modifizieren. Mit Kenntnis der oben beschriebenen Bit-Operationen sollten Sie nachvollziehen können, wenn eine dieser Bit-Operationen nun bitweise auf eine Gruppe von Bytes zeitgleich stattfindet.

Operiert man auf zwei Bytes (oder auf zwei Bytegruppen), so verarbeitet dabei dabei jeweils Bits, die in diesen Bytes (bzw. in diesen beiden Bytegruppen) an gleicher Bit-Positionen stehen, so werden also

- die beiden Bits an Bit-Position 0 miteinander verarbeitet;
- die beiden Bits an Bit-Position 1 miteinander verarbeitet;
- die beiden Bits an Bit-Position 2 miteinander verarbeitet;
- die beiden Bits an Bit-Position 3 miteinander verarbeitet;
- usw.

Programmiersprachen, die lauffähige Programme für eine bestimmte Computerplattform hervorbringen, müssen deren Hardwaregegebenheiten berücksichtigen. Ihre Bit-Operationen beziehen sich daher auf Bytes bzw. Bytegruppen. In der Regel stellen Programmiersprachen wenigstens folgende Operationen bereit:

- Bitweise Operationen:
  - AND
  - OR
  - XOR

EVO PDF Tools Demo

- Für Zahlen-Vergleiche:
  - $=$ (*Gleich*)
  - $<$ (*Kleiner-Als*, für vorzeichenbehaftete Integer)
  - $\mathrm{U}<$ (*Kleiner-Als*, für vorzeichenlose Integer)

Mittels diesen wenigen Operatoren lassen sich alle weiteren bitweise Operatoren durch geschickte Kombination erstellen. Wir werden das exemplarisch für die Programmiersprache Forth vorstellen. Was Sie dabei lernen, können Sie hinterher auf andere Programmiersprachen übertragen.

Für folgenden Betrachtungen gehen wir davon aus, daß die genannten Operationen AND, OR, XOR, $=$, $<$ und $\mathrm{U}<$ auf sogenannte **Zellen** operieren. Jede Zelle ist dabei eine Bytegruppe aus 2 Byte zu je 8 Bit. Eine Zelle ist somit 16 Bit ($= 2 * 8$ Bit) groß. Eine Zelle ist zugleich der Speicherraum in dem einfache Ganzzahlen (Integer) abgelegt und behandelt werden.

Die Struktur von Integer ist jedoch etwas komplexer als man zunächst vermuten mag. So erhält man beispielsweise von einer positiven Integer nicht ihr negatives Pendant, indem man

- einfach sämtliche Bits der positiven Zahl invertiert, oder
- ein Bit als Vorzeichen-Bit festlegt und lediglich dieses invertiert.

Tatsächlich ist der Vorgang etwas komplexer. Das zeigt bereits ein wechselseitiger Vergleich der Bitmuster folgender Integer:

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Die Zahl **0**

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Die negative Zahl **-1**

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Die positive Zahl **1**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	Die negative Zahl **-2**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	Die positive Zahl **2**
0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	Die positive Zahl **147**
1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	Die negative Zahl **-147**

Um das Vorzeichen einer Integer zu negieren (d.h. gegen das jeweils andere Vorzeichen auszutauschen) werden wir weiter unten eine Routine namens NEGATE anlegen, die diese Aufgabe erledigt.

Wir merken uns: Das Negieren einer Integer ergibt ein anderes Bitmuster als das Invertieren all Ihrer Bits.

Bit-Operationen auf Zellbreite

Unsere Betrachtungen für »*Bit-Operationen auf Zellbreite*« werden wir in drei alternativen Codemodellen entwickeln, und zwar zweimal auf Basis von Pseudocode und einmal auf Basis von echtem Programmcode. Zu Vergleichs- und Verständniszwecken werden wir diese drei Codemodelle für jede einzelne Operation in ein übersichtliches Tabellenschema eintragen:

Ⓐ **Pseudocode**	Ⓑ **Pseudocode**
Dieser Tabellenbereich enthält infix- / funktional-orientierten Pseudocode.	Dieser Tabellenbereich enthält postfix-orientierten Pseudocode.
Ⓒ **Programmcode**	
Dieser Tabellenbereich enthält in Forth geschriebenen Programmcode.	

EVO PDF Tools Demo

Somit tragen wir

- in das Tabellenfeld »A« einen infix- bzw. funktional-orientierten Pseudocode ein;
- in das Tabellenfeld »B« einen postfix-orientierten Pseudocode ein; und
- in das Tabellenfeld »C« einen in Forth formulierten realen Programmcode ein.

Eine Besonderheit der Programmiersprache Forth ist, daß sie die Postfix-Schreibweise nutzt, also ähnlich dem Pseudocode in Tabellenfeld »B«, dabei jedoch etwas schlichter und kompakter auftritt.

Eine weitere Besonderheit von Forth ist, daß die Namen der in ihr erstellten Routinen sämtliche druckbare Schriftzeichen enthalten dürfen, die kein Leerzeichen sind. Erlaubte Namenszeichen sind somit beispielsweise auch: `/` , `<` , `>` , `=` , `.` , `:` , `;` , `-` , `+` oder `@` .

Wo immer solch ungewöhnliche Zeichen in Namen von Forthwörtern auftauchen, handelt es sich nicht um Syntax- oder Operationszeichen, sondern um Schriftzeichen, die, wie wie Buchstaben und auch Ziffern, den Namen des Forthwortes darstellen; jedoch werden einige dieser ungewöhnlichen Zeichen von Forth-Programmierern bevorzugt hinsichtlich bestimmter Bedeutungszusammenhänge genutzt. Dies ermöglicht Forth-Programmierern ein verbessertes Verständnis und eine symbolisch unterstützte Lesbarkeit ihres Programmcodes. Doch brauchen wir uns darüber keine Gedanken machen, denn wir nutzen den Forth-Code wegen seiner formalen Einfachheit und Kompaktheit. Wir kommen so zu einem raschen Verständnis aller mittels Forth formulierter »Bit-Operationen auf Zellbreite«.

Für das Verständnis der »Bit-Operationen auf Zellbreite« sind Sie jedoch nicht dem Forth-Code in Tabellenfeld »C« ausgeliefert, sondern können sich alternativ den Pseudocode-Schreibweisen der Tabellenfelder »A« und »B« zuwenden.

Mit den in den Tabellenfelder »A« bis »C« präsentierten Codezeilen sollten Sie der Lage sein äquivalente Routinen in einer Programmiersprache Ihrer Wahl zu schreiben und lauffähig zu machen. Dabei sollten Sie jedoch im Auge

behalten, daß die hier formulierten Pseudocodezeilen und Forth-Codezeilen durchgehend Bitmuster auf Zellenbreite behandeln und daher typenlos sind. Wie die beteiligen Bitfolgen dabei modifizieren bzw. modifiziert werden, liegt ausschließlich daran, welche Operationen dabei innerhalb der Routinen durchgeführt werden. Am Ende werden eine (ggf. auch mehrere) zellbreite Bitfolgen in einer definierten Reihenfolge als Ergebnis zurückgegeben. Um Ihnen jedoch eine Orientierung zu geben, wie die beteiligten Bitfolgen vorzugsweise anzusehen sind, nutzen wir in allen Pseudocodezeilen einfache Variablen-Schreibweise. Diese legt jedoch keine Datentypen fest, sondern gibt lediglich Hinweise, wie die beteiligten Bitfolgen vorzugsweise zu deuten sind. Für Variablen und Parameter nutzen wir folgende Schreibweise:

- **n** : vorzeichenbehaftete Integer
- **+n** : vorzeichenbehaftete positive Integer
- **u** : vorzeichenlose Integer
- **f** : Boolescher Wahrheitswert (Flag)

Ein letzter Hinweis: Falls in Tabellenfeld »C« irgendwo der Hinweis »`//vordefiniert///`« notiert sein sollte, weist das darauf hin, daß diese Operation und ihr Forthname bereits im Forth-System implementiert ist. Es handelt sich dann um eine grundlegende und herstellerseitig vorimplementierte Operation des Forth-Systems.

Und nun zu den »Bit-Operationen auf Zellbreite« für Computer mit einer Zellgröße von 2 Byte zu 8 Bit je Zelle:

BIT/BYTE **(-- n)**
BitPerByte (-- n)

> Liefert mit *n* die Anzahl der Bits je Byte.

```
def BitPerByte():( n )
{
  set n :=  8 ;
  return n ;
}
```

```
def BitPerByte    ( -- n )
{
  set n :=  8 ;
  return n ;
}
```

```
8  CONSTANT BIT/BYTE                    ( -- n )
```

EVO PDF Tools Demo

Die runden Klammern `(` und `)` am Ende des Forth-Codes legen einen Kommentartext fest. Als Kommentartext gilt alles, was zwischen den beiden Klammern notiert ist.

BYTE/CELL **(-- n)**
BytePerCell (-- n)

> Liefert mit *n* die Anzahl der Bytes je Zelle.

```
def BytePerCell():( n )
{
  set n :=  2 ;
  return n ;
}
```

```
def BytePerCell    ( -- n )
{
  set n :=  2 ;
  return n ;
}
```

```
2 CONSTANT BYTE/CELL                    ( -- n )
```

ALL.ONE **(-- u)**
AllOne (-- u)

> Liefert mit *u* eine Zahl, bei der alle Bits 1 sind.
>
> Schema:

MSB							LSB
7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1

Alle Bit sind 1

```
def AllOne():( u )
{
  set u :=  -1 ;
  return u ;
}
```

```
def AllOne    ( -- u )
{
  set u :=  -1 ;
  return u ;
}
```

```
-1  CONSTANT ALL.ONE                                    ( -- u )
```

ALL.ZERO **(-- u)**

AllZero (-- u)

Liefert mit *u* eine Zahl, bei der alle Bits 0 sind.

Schema:

MSB							LSB
7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

Alle Bit sind 0

```
def AllZero():( u )
{
  set u :=  0 ;
  return u ;
}
```

```
def AllZero    ( -- u )
{
  set u :=  0 ;
  return u ;
}
```

```
0  CONSTANT ALL.ZERO                                   ( -- u )
```

EVO PDF Tools Demo

AND **(u1 u2 -- u3)**

And (u1 u2 -- u3)

Liefert mit *u3* das Ergebnis der bitweisen AND-Verknüpfung von *u1* und *u2* .

```
def And( u1 , u2 ):( u3 )
{
  set u3 :=  ( u1 ∧ u2 ) ;
  return u3 ;
}
```

```
def And    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 ∧ ;
  return u3 ;
}
```

```
( AND       //vordefiniert////                    { u1 u2 -- u3 )
```

OR **(u1 u2 -- u3)**

Or (u1 u2 -- u3)

Liefert mit *u3* das Ergebnis der bitweisen OR-Verknüpfung von *u1* und *u2* .

```
def Or( u1 , u2 ):( u3 )
{
  set u3 :=  ( u1 ∨ u2 ) ;
  return u3 ;
}
```

```
def Or    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 ∨ ;
  return u3 ;
}
```

```
( OR        //vordefiniert////                    { u1 u2 -- u3 )
```

XOR **(u1 u2 -- u3)**

Xor (u1 u2 -- u3)

Liefert mit *u3* da Ergebnis der bitweisen XOR-Verknüpfung von *u1* und *u2* .

```
def Xor( u1 , u2 ):( u3 )
{
  set u3 :=  ( u1 V u2 ) ;
  return u3 ;
}
```

```
def Xor    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 V ;
  return u3 ;
}
```

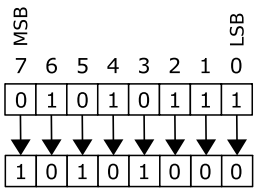
```
( XOR       //vordefiniert////                    { u1 u2 -- u3 )
```

INVERT **(u -- u')**

Invert (u -- u')

Invertiert alle Bit-Zustände in *u* ; d.h. jede 0 wird zu 1, und jede 1 wird zu 0. Das Ergebnis wird als *u'* zurückgegeben.

Schema:



Invertieren aller Bits

```
def Invert( u ):( u' )
{
  set u' :=  Xor( u , AllOne() ) ;
  return u' ;
}
```

```
def Invert    ( u -- u' )
{
  set u' :=  u AllOne Xor ;
  return u' ;
}
```

```
: INVERT       ALL.ONE XOR ;                    ( u -- u' )
```

EVO PDF Tools Demo

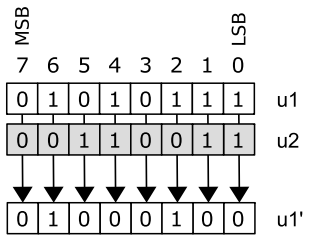
Das Forthwort `:` leitet die Erstellung einer Routine (d.h. eines neuen Forthwortes) ein. Es folgt ein Leerzeichen und anschließend der Name des neuen Forthwortes. Alles danach folgende sieht Forth als Anweisungen an und kompiliert diese in die neue Routine; allerdings werden Kommentare nicht hineinkompiliert. Sollte Forth auf einen Fehler oder eine ihm unbekannte Anweisung treffen, bricht es den Kompilervorgang sofort ab. Erreicht der Forth-Kompiler schließlich das Forthwort `;` wird der Kompilervorgang abgeschlossen und das neue Forthwort dem Forth-System augenblicklich verfügbar gemacht.

INHIBIT **(u1 u2 -- u1')**

Inhibit (u1 u2 -- u1')

Übernimmt *u1* , doch werden dabei alle Bit-Positionen auf 0 gesetzt, die in *u2* den Wert 1 haben. Das Ergebnis wird als *u1'* zurückgegeben. *u2* legt also fest, welche Bit-Positionen von *u1* durchgelassen oder gehemmt (blockiert / degradiert) werden.

Schema:



Hemmen ausgewählter Bits

```
def Inhibit( u1 , u2 ):( u1' )
{
  set u1' :=  And( u1 , Invert( u2 )) ;
```

```
def Inhibit    ( u1 u2 -- u1' )
{
  set u1' :=  u1 u2 Invert And ;
```

```
  return u1' ;
}
```

```
: INHIBIT      INVERT AND ;                    ( u1 u2 -- u1' )
```

```
  return u1' ;
}
```

NAND **(u1 u2 -- u3)**

Nand (u1 u2 -- u3)

Liefert mit *u3* das Ergebnis der bitweisen NAND-Verknüpfung von *u1* und *u2* .

```
def Nand( u1 , u2 ):( u3 )
{
  set u3 :=  Invert( And( u1 , u2 )) ;
  return u3 ;
}
```

```
def Nand    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 And Invert ;
  return u3 ;
}
```

```
: NAND         AND INVERT ;                    ( u1 u2 -- u3 )
```

NOR **(u1 u2 -- u3)**

Nor (u1 u2 -- u3)

Liefert mit *u3* das Ergebnis der bitweisen NOR-Verknüpfung von *u1* und *u2* .

```
def Nor( u1 , u2 ):( u3 )
{
  set u3 :=  Invert( Or( u1 , u2 )) ;
  return u3 ;
}
```

```
def Nor    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 Or Invert ;
  return u3 ;
}
```

```
: NOR          OR INVERT ;                     ( u1 u2 -- u3 )
```

EVO PDF Tools Demo

XNOR **(u1 u2 -- u3)**

Xnor (u1 u2 -- u3)

Liefert mit *u3* das Ergebnis der bitweisen XNOR-Verknüpfung von *u1* und *u2* .

```
def Xnor( u1 , u2 ):( u3 )
{
  set u3 :=  Invert( Xor( u1 , u2 )) ;
  return u3 ;
}
```

```
def Xnor    ( u1 u2 -- u3 )
{
  set u3 :=  u1 u2 Xor Invert ;
  return u3 ;
}
```

```
: XNOR         XOR INVERT ;                    ( u1 u2 -- u3 )
```

TRUE **(-- f)**

True (-- f)

Liefert ein Flag mit dem definierten Booleschen Wahrheitswert *»true«*.

Schema:

MSB							LSB
7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1

»true«

```
def True():( f )
{
```

```
def True    ( -- f )
{
```

```
  set f :=  AllOne() ;
  return f ;
}
```

```
  set f :=  AllOne ;
  return f ;
}
```

```
ALL.ONE  CONSTANT TRUE                    ( -- TRUE )
```

FALSE **(-- f)**
False (-- f)

> Liefert ein Flag mit dem definierten Booleschen Wahrheitswert *»false«*.
>
> Schema:

MSB							LSB
7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

»false«

```
def False():( f )
{
  set f :=  AllZero() ;
  return f ;
}
```

```
def False    ( -- f )
{
  set f :=  AllZero ;
  return f ;
}
```

```
ALL.ZERO  CONSTANT FALSE                  ( -- FALSE )
```

NOT **(u -- f)**
Not (u -- f)

EVO PDF Tools Demo

> Liest *n* als Flag und liefert dessen negierten Wahrheitswert als konsolidiertes Boolesches Flag (*true* oder *false*).

```
def Not( u ):( f )
{
  set f :=  ( u  = False() ) ;
  return f ;
}
```

```
def Not    ( u -- f )
{
  set f :=  u False  = ;
  return f ;
}
```

```
: NOT          FALSE = ;                  ( u -- f )
```

BOOL **(u -- f)**
Bool (u -- f)

> Liest *n* als Flag und liefert dessen konsolidierten Wahrheitswert als Boolesches Flag (*true* oder *false*).

```
def Bool( u ):( f )
{
  set f :=  Not( Not( u )) ;
  return f ;
}
```

```
def Bool    ( u -- f )
{
  set f :=  u Not Not ;
  return f ;
}
```

```
: BOOL         NOT NOT ;                  ( u -- f )
```

< **(n1 n2 -- f)**
LessThan (n1 n2 -- f)

> Das Flag *f* ist *»true«*, wenn *n1* kleiner als *n2* ist; anderenfalls ist es *»false«*.

```
def LessThan( n1 , n2 ):( f )
{
  set f :=  ( n1 < n2 ) ;
  return f ;
}
```

```
def LessThan    ( n1 n2 -- f )
{
  set f :=  n1 n2 < ;
  return f ;
}
```

```
( <            //vordefiniert////                { n1 n2 -- f )
```

> **(n1 n2 -- f)**

LargerThan (n1 n2 -- f)

Das Flag *f* ist *»true«*, wenn *n1* größer als *n2* ist; anderenfalls ist es *»false«*.

```
def LargerThan( n1 , n2 ):( f )
{
  set f :=  LessThan( n2 , u1 ) ;
  return f ;
}
```

```
def LargerThan    ( n1 n2 -- f )
{
  set f :=  n2 n1 LessThan ;
  return f ;
}
```

```
: >            SWAP < ;                          ( n1 n2 -- f )
```

= **(n1 n2 -- f)**

Equal (n1 n2 -- f)

Das Flag *f* ist *»true«*, wenn *n1* gleich *n2* ist; anderenfalls ist es *»false«*.

```
def Equal( n1 , n2 ):( f )
{
  set f :=  ( n1 = n2 ) ;
  return f ;
}
```

```
def Equal    ( n1 n2 -- f )
{
  set f :=  n1 n2 = ;
  return f ;
}
```

EVO PDF Tools Demo

```
( =            //vordefiniert////                { n1 n2 -- f )
```

<> **(n1 n2 -- f)**

NotEqual (n1 n2 -- f)

Das Flag *f* ist *»true«*, wenn *n1* ungleich *n2* ist; anderenfalls ist es *»false«*.

```
def NotEqual( n1 , n2 ):( f )
{
  set f :=  Not( Equal( n1 , n2 )) ;
  return f ;
}
```

```
def NotEqual    ( n1 n2 -- f )
{
  set f :=  n1 n2 Equal Not ;
  return f ;
}
```

```
: <>           = NOT ;                           ( n1 n2 -- f )
```

U< **(u1 u2 -- f)**

ULessThan (u1 u2 -- f)

Das Flag *f* ist *»true«*, wenn *u1* kleiner als *u2* ist; anderenfalls ist es *»false«*.

```
def ULessThan( u1 , u2 ):( f )
{
  set f :=  ( u1 < u2 ) ;
  return f ;
}
```

```
def ULessThan    ( u1 u2 -- f )
{
  set f :=  u1 u2 < ;
  return f ;
}
```

```
( U<           //vordefiniert////                { n1 n2 -- f )
```

U> **(u1 u2 -- f)**
ULargerThan (u1 u2 -- f)

> Das Flag *f* ist *»true«*, wenn *u1* größer als *u2* ist; anderenfalls ist es *»false«*.

```
def ULargerThan( u1 , u2 ):( f )
{
  set f :=  ULessThan( u2 , u1 ) ;
  return f ;
}
```

```
def ULargerThan    ( u1 u2 -- f )
{
  set f :=  u2 u1 ULessThan ;
  return f ;
}
```

```
: U>          SWAP U< ;                    ( u1 u2 -- f )
```

U= **(u1 u2 -- f)**
UEqual (u1 u2 -- f)

> Das Flag *f* ist *»true«*, wenn *u1* gleich *u2* ist; anderenfalls ist es *»false«*.

```
def UEqual( u1 , u2 ):( f )
{
  set f :=  Equal( u1 , u2 ) ;
  return f ;
}
```

```
def UEqual    ( u1 u2 -- f )
{
  set f :=  u1 u2 Equal ;
  return f ;
}
```

```
: U=          = ;                          ( u1 u2 -- f )
```

U<> **(u1 u2 -- f)**
UNotEqual (u1 u2 -- f)

EVO PDF Tools Demo

> Das Flag *f* ist *»true«*, wenn *u1* ungleich *u2* ist; anderenfalls ist es *»false«*.

```
def UNotEqual( u1 , u2 ):( f )
{
  set f :=  NotEqual( u1 , u2 ) ;
  return f ;
}
```

```
def UNotEqual    ( u1 u2 -- f )
{
  set f :=  u1 u2 NotEqual ;
  return f ;
}
```

```
: U<>         <> ;                         ( u1 u2 -- f )
```

0= **(n -- f)**
EqualZero (n -- f)

> Das Flag *f* ist *»true«*, wenn *n* gleich 0 ist; anderenfalls ist es *»false«*.

```
def EqualZero( n ):( f )
{
  set f :=  Equal( n , 0 ) ;
  return f ;
}
```

```
def EqualZero    ( n -- f )
{
  set f :=  n 0 Equal ;
  return f ;
}
```

```
: 0=          0 = ;                        ( n -- f )
```

0<> **(n -- f)**
NotEqualZero (n -- f)

Das Flag *f* ist *»true«*, wenn *n* ungleich 0 ist; anderenfalls ist es *»false«*.

```
def NotEqualZero( n ):( f )
{
  set f :=  NotEqual( n , 0 ) ;
  return f ;
}
```

```
def NotEqualZero    ( n -- f )
{
  set f :=  n 0 NotEqual ;
  return f ;
}
```

```
: 0<>          0 = ;                                    ( n -- f )
```

0> **(n -- f)**
LargerThanZero (n -- f)

Das Flag *f* ist *»true«*, wenn *n* größer als 0 ist; anderenfalls ist es *»false«*.

```
def LargerThanZero( n ):( f )
{
  set f :=  LargerThan( n , 0 ) ;
  return f ;
}
```

```
def LargerThanZero    ( n -- f )
{
  set f :=  n 0 LargerThan ;
  return f ;
}
```

```
: 0>           0 > ;                                    ( n -- f )
```

0< **(n -- f)**
LessThanZero (n -- f)

Das Flag *f* ist *»true«*, wenn *n* kleiner als 0 ist; anderenfalls ist es *»false«*.

```
def LessThanZero( n ):( f )
{
  set f :=  LessThan( n , 0 ) ;
  return f ;
}
```

```
def LessThanZero    ( n -- f )
{
  set f :=  n 0 LessThan ;
  return f ;
}
```

EVO PDF Tools Demo

```
: 0<           0 < ;                                    ( n -- f )
```

POSITIVE? **(n -- f)**
IsPositive (n -- f)

Das Flag *f* ist *»true«*, wenn *n* eine positive Integer ist; anderenfalls ist es *»false«*.

```
def IsPositiv( n ):( f )
{
  set f :=  LargerThanZero( n ) ;
  return f ;
}
```

```
def IsPositiv    ( n -- f )
{
  set f :=  n LargerThanZero ;
  return f ;
}
```

```
: POSITIV?     0> ;                                     ( n -- f )
```

NEGATIVE? **(n -- f)**
IsNegative (n -- f)

Das Flag *f* ist *»true«*, wenn *n* eine negative Integer ist; anderenfalls ist es *»false«*.

```
def IsNegative( n ):( f )
{
  set f :=  LessThanZero( n ) ;
  return f ;
}
```

```
def IsNegative    ( n -- f )
{
  set f :=  n LessThanZero ;
  return f ;
}
```

```
: NEGATIVE?    0< ;                                     ( n -- f )
```

NEGATE **(n -- n')**
Negate (n -- n')

Ändert das Vorzeichen der Zahl *n* (jedoch nicht deren Betrag) und gibt das Ergebnis als *n'* zurück.

- Ist *n* eine positive Integer, so gibt NEGATE deren negatives Pendant zurück.
- Ist *n* eine negative Integer, so gibt NEGATE deren positives Pendant zurück.

(Diese Operation entspricht damit einer Multiplikation von *n* mit -1 .)

```
def Negate( n ):( n' )
{
  set n' :=  Invert( n ) + 1 ;
  return f ;
}
```

```
def Negate    ( n -- n' )
{
  set f :=  n Invert 1 + ;
  return f ;
}
```

```
: NEGATE       INVERT 1+ ;                          ( n -- n' )
```

BYTE>BIT **(+n1 -- n2)**
ByteToBit (+n1 -- n2)

Rechnet *+n1* Bytes in Bits um Das Ergebnis wird als *n2* zurückgegeben.

```
def ByteToBit( n1 ):( n2 )
{
  set n2 :=  Abs( n1 ) * BitPerByte() ;
  return n2 ;
}
```

```
def ByteToBit    ( n1 -- n2 )
{
  set n2 :=  n1 Abs BitPerByte * ;
  return n2 ;
}
```

```
: BYTE>BIT      ABS BIT/BYTE * ;                    ( +n1 -- n2 )
```

EVO-PDF Tools Demo

BIT>BYTE **(+n1 -- n2)**
BitToByte (+n1 -- n2)

Rechnet *+n1* Bits in Bytes um. Das Ergebnis wird als *n2* zurückgegeben.

```
def BitToByte( n1 ):( n2 )
{
  set n2 :=  Negate(
                 Abs( n1 ) /floor Negate( BitPerByte() )
               ) ;
  return n2 ;
}
```

```
def BitToByte    ( n1 -- n2 )
{
  set n2 :=  n1 Abs
             BitPerByte Negate
             /floor Negate ;
  return n2 ;
}
```

```
: BIT>BYTE       ABS BIT/BYTE NEGATE / NEGATE ;      ( n1 -- n2 )
```

BIT/CELL **(-- n)**
BitPerCell (-- n)

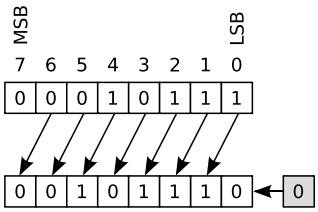
Liefert mit *n* die Anzahl der Bits je Zelle.

```
def BitPerCell():( n )
{
  set n :=  ByteToBit( BytePerCell() ) ;
  return n ;
}
```

```
def BitPerCell    ( -- n )
{
  set n :=  BytePerCell ByteToBit ;
  return n ;
}
```

```
BYTE/CELL BYTE>BIT    CONSTANT BIT/CELL                ( -- n )
```

<< **(u -- u')**
LeftShift (u -- u')

> Logische Links-Verschiebung der Bits in *u* um eine Bit-Position. Dabei verfällt das MSB von *u* . Das Ergebnis wird als *u'* zurückgegeben.
>
> Schema:
>
> 
>
>
> Logische Links-Verschiebung
>
> ```
> def LeftShift( u ):( u' )
> {
>  set u' :=  u * 2 ;
>  return u' ;
> }
> ```
>
> ```
> def LeftShift    ( u -- u' )
> {
>  set u' :=  u 2 * ;
>  return u' ;
> }
> ```
>
> ```
> : <<       2* ;                                    ( u -- u' )
> ```

Die logische und die arithmetische Links-Verschiebung sind sind dieselbe Bit-Operation. Somit deckt die logische Links-Verschiebung beide ab, und die arithmetische Links-Verschiebung braucht nicht gesondert programmiert werden.

EVO PDF Tools Demo

<<< **(u +n -- u')**
MultiLeftShift (u +n -- u')

> Logische Links-Verschiebung der Bits in *u* um *+n* Bit-Positionen. Dabei verfallen u.U. mehrere MSBs von *u* . Das Ergebnis wird als *u'* zurückgegeben.
>
> ```
> def MultiLeftShift( u , n ):( u' )
> {
>  set u' :=  u ;
>  LOOP( 0 , Abs( n ) - 1 )
>  {
>  set u' :=  LeftShift( u' ) ;
>  };
>  return u' ;
> }
> ```
>
> ```
> def MultiLeftShift    ( u n -- u' )
> {
>  set u' :=  u ;
>  0
>  n Abs 1 -
>  LOOP{
>  set u' :=  u' LeftShift ;
>  };
>  return u' ;
> }
> ```
>
> ```
> : <<<      ABS 0 ?DO  <<  LOOP ;                    ( u +n -- u' )
> ```

Die **LOOP**-Struktur (auch »Zählschleife« genannt) der Pseudocodes in den Tabellenfeldern »A« und »B« nutzt jeweils zwei Parameter. Der erste Parameter ist der Startwert, mit dem die Zählschleife das erste mal durchlaufen wird; der zweite Parameter ist der Endwert, mit dem die Zählschleife das letzte mal durchlaufen und dann beendet wird. Beginnt eine Zählschleife mit dem Startwert 0, so ist der Endwert häufig um 1 zu mindern, um die gewünschte Anzahl an Schleifendurchläufen zu erreichen. Mit jedem Schleifendurchlauf werden die zwischen den geschweiften Klammern der LOOP-Struktur aufgeführten Anweisungen ausgeführt.

Der Forth-Code in Tabellenfeldern »C« verzichtet bei Zählschleifen auf geschweifte Klammern. Statt dessen wird die Zählschleifen mit dem Forthwort **?DO** eingeleitet, und mit dem Forthwort **LOOP** abgeschlossen. Mit jedem Schleifendurchlauf werden die dazwischen stehenden Anweisung der Reihe nach ausgeführt. Eine Erleichterung bei Forth ist, daß die Endwerte nicht um 1 reduziert werden müssen, um eine passende Anzahl an Schleifendurchläufen zu erhalten, sofern der Startwert einer Zählschleife 0 ist. Forthwörter sind so beschaffen, daß ihre Nutzung für Forth-Programmierer möglichst einfach und verständlich ist.

LSB.ONE **(-- u)**
LsbOne (-- u)

Liefert mit *u* eine Zahl, bei der nur das LSB auf 1 gesetzt ist, alle übrigen Bits sind 0.

Schema:

MSB							LSB
7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

Nur LSB ist 1

```
def LsbOne():( u )
{
  set n :=  1 ;
  return n ;
}
```

```
def LsbOne    ( -- u )
{
  set n :=  1 ;
  return n ;
}
```

```
1  CONSTANT LSB.ONE                                    ( -- u )
```

MSB.ONE **(-- u)**
MsbOne (-- u)

Liefert mit *u* eine Zahl, bei der nur das MSB auf 1 gesetzt ist, alle übrigen Bits sind 0.

Schema:

MSB							LSB
7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

EVO PDF Tools Demo

Nur MSB ist 1

```
def MsbOne():( u )
{
  set u :=  MultiLeftShift(
                LsbOne() ,
                BitPerCell() - 1
            ) ;
  return u ;
}
```

```
def MsbOne    ( -- u )
{
  set u :=  LsbOne
            BitPerCell 1 -
            MultiLeftShift ;
  return u ;
}
```

```
LSB.ONE BIT/CELL 1- <<<   CONSTANT MSB.ONE              ( -- u )
```

Das LSB wird auf 1 gesetzt und anschließend dessen Status soweit nach links verschoben, daß dieser ins MSB gelangt. Die Zahl der Bit-Positionen, um die der Statuswert des LSB dabei nach links verschoben wird, ist die Anzahl der Bytes pro Zelle gemindert um 1.

INT.MAX **(-- +n)**
IntMax (-- +n)

Liefert mit *+n* den maximale vorzeichenbehaftete positive Integer. Diese Zahl ist dadurch gekennzeichnet, daß ihr MSB den Wert 0 hat, und die übrigen Bits 1 sind.

Schema:

MSB							LSB
7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1

größter positiver Integerwert

```
def IntMax():( n )
{
  set n :=  Invert( MsbOne() ) ;
  return n ;
}
```

```
def IntMax    ( -- n )
{
  set n :=  MsbOne Invert ;
  return n ;
}
```

```
MSB.ONE INVERT  CONSTANT INT.MAX                    ( -- +n )
```

Dieses Vorgehen behandelt die Bitfolge *MSB.ONE* so, daß daraus die Bitfolge der größten positiven Integer entsteht. Grundsätzlich kann jede Bitfolge als Zahl angesehen werden. So repräsentiert die Bitfolge *MSB.ONE* zugleich die kleinste vorzeichenlose Integer, die außerhalb (d.h. oberhalb) des Wertebereichs der vorzeichenbehafteten Integer liegt. Mindert man also *MSB.ONE* um 1, so erhält man den Zahlenwert der größten vorzeichenbehafteten positiven Integer. Diese kann also auch berechnet werden. Der entsprechende Forth-Code lautet:

```
MSB.ONE 1- CONSTANT INT.MAX
```

MSB **(u -- u')**
Msb (u -- u')

Liefert nur das MSB der Zahl *u* ; die übrigen Bit von *u'* sind 0. Das Ergebnis wird als *u'* zurückgegeben.

```
def Msb( u ):( u' )
{
  set u' :=  And( u , MsbOne() ) ;
  return u' ;
}
```

```
def Msb    ( u -- u' )
{
  set u' :=  u MsbOne And ;
  return u' ;
}
```

```
: MSB          MSB.ONE AND ;                    ( u -- u' )
```

LSB **(u -- u')**
Lsb (u -- u')

EVO PDF Tools Demo

Liefert nur das LSB der Zahl *u* ; die übrigen Bit von *u'* sind 0. Das Ergebnis wird als *u'* zurückgegeben.

```
def Lsb( u ):( u' )
{
  set u' :=  And( u , LsbOne() ) ;
  return u' ;
}
```

```
def Lsb    ( u -- u' )
{
  set u' :=  u LsbOne And ;
  return u' ;
}
```

```
: LSB          LSB.ONE AND ;                    ( u -- u' )
```

>MSB **(f -- u)**
ToMsb (f -- u)

Liest das Flag *f* und trägt dessen Zustand im MSB von *u'* ein; die übrigen Bits von *u'* sind 0.

```
def ToMsb( f ):( u )
{
  set u :=  Msb( Bool( f )) ;
  return u ;
}
```

```
def ToMsb    ( f -- u )
{
  set u :=  f Bool Msb ;
  return u ;
}
```

```
: >MSB         BOOL MSB ;                       ( f -- u )
```

>LSB **(f -- u)**
ToLsb (f -- u)

Liest das Flag *f* und trägt dessen Zustandswert im LSB von *u'* ein; die übrigen Bits von *u'* sind 0.

```
def ToLsb( f ):( u )
{
  set u :=  Lsb( Bool( f )) ;
  return u ;
}
```

```
def ToLsb    ( f -- u )
{
  set u :=  f Bool Lsb ;
  return u ;
}
```

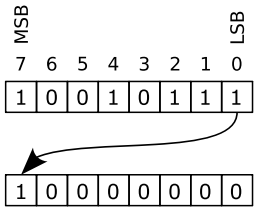
```
: >LSB          BOOL LSB ;                          ( f -- u )
```

LSB>MSB **(u -- u')**

LsbToMsb (u -- u')

Macht das LSB von *u* zum MSB von *u'*; die übrigen Bits von *u'* werden auf 0 gesetzt.

Schema:



LSB>MSB

```
def LsbToMsb( u ):( u' )
{
  set u' :=  ToMsb( Lsb( u )) ;
  return u' ;
}
```

```
def LsbToMsb    ( u -- u' )
{
  set u' :=  u Lsb ToMsb ;
  return u' ;
}
```

```
: LSB>MSB       LSB >MSB ;                          ( u -- u' )
```

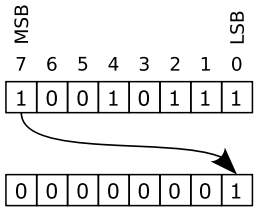
EVO PDF Tools Demo

MSB>LSB **(u -- u')**

MsbToLsb (u -- u')

Macht das MSB von *u* zum LSB von *u'*; die übrigen Bits von *u'* werden auf 0 gesetzt.

Schema:



MSB>LSB

```
def MsbToLsb( u ):( u' )
{
  set u' :=  ToLsb( Msb( u )) ;
  return u' ;
}
```

```
def MsbToLsb    ( u -- u' )
{
  set u' :=  u Msb ToLsb ;
  return u' ;
}
```

```
: MSB>LSB       MSB >LSB ;                          ( u -- u' )
```

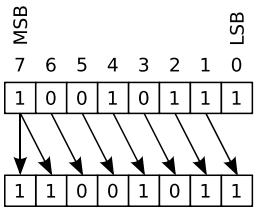
A>> **(n -- n')**

ArithRightShift (n -- n')

Arithmetische Rechts-Verschiebung der Bits von *n* um eine Bit-Position. Das LSB verfällt. Das Ergebnis wird

als *n'* zurückgegeben.

Schema:



Arithmetische Rechts-Verschiebung

```
def ArithRightShift( n )( n' )
{
  set n' :=  n / 2 ;
  return n' ;
}
```

```
def ArithRightShift    ( n -- n' )
{
  set n' :=  n 2 / ;
  return n' ;
}
```

```
: A>>          2/ ;                                ( n -- n' )
```

Bei der arithmetischen Rechts-Verschiebung bleibt das Vorzeichen-Bit (das MSB) erhalten.

A>>> **(n1 +n2 -- n1')**
MultiArithRightShift (n1 +n2 -- n1')

Arithmetische Rechts-Verschiebung der Bits von *n1* um *+n2* Bit-Positionen. Dabei verfallen u.U. mehrere LSBs von *n1* . Das Ergebnis wird als *n1'* zurückgegeben.

```
def MultiArithRightShift( n1 , n2 ):( n1' )
{
  set n1' :=  n1 ;
  LOOP( 0 , Abs( n2 ) - 1 )
  {
    set n1' :=  ArithRightShift( n1' ) ;
  };
  return n1' ;
}
```

```
def MultiArithRightShift    ( n1 n2 -- n1' )
{
  set n1' :=  n1 ;
  0
  n2 Abs 1 -
  LOOP{
    set n1' :=  n1' ArithRightShift ;
  };
  return n1' ;
}
```

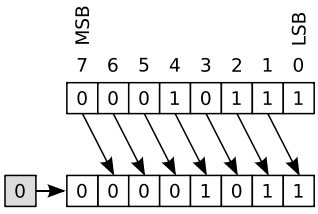
EVO PDF Tools Demo

```
: A>>>         ABS 0 ?DO  A>>  LOOP ;          ( n1 +n2 -- n1' )
```

>> **(u -- u')**
RightShift (u -- u')

Logische Rechts-Verschiebung der Bits von *u* um eine Bit-Position. Das LSB verfällt. Das Ergebnis wird als *u'* zurückgegeben.

Schema:



Logische Rechts-Verschiebung

```
def RightShift( u ):( u' )
{
  set u' :=  And( u / 2 , IntMax() ) ;
  return u' ;
}
```

```
def RightShift    ( u -- u' )
{
  set u' :=  u 2 / IntMax And ;
  return u' ;
}
```

```
: >>           2/ INT.MAX AND ;                    ( u -- u' )
```

Bei der logischen Rechts-Verschiebung wird das Vorzeichen-Bit (das MSB) mitverschoben.

>>> **(u +n -- u')**
MultiRightShift (u +n -- u')

Logische Rechts-Verschiebung der Bits von *u* um *+n* Bit-Positionen. Dabei verfallen u.U. mehrere LSBs von *u* . Das Ergebnis wird als *u'* zurückgegeben.

```
def MultiRightShift( u , n ):( u' )
{
  set u' :=  u ;
  LOOP( 0 , Abs( n ) - 1 )
  {
    set u' :=  RightShift( u' ) ;
  };
  return u1' ;
}
```

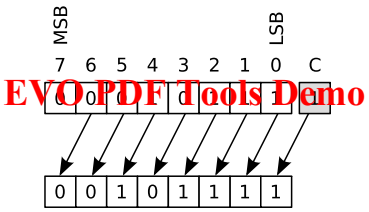
```
def MultiRightShift    ( u n -- u' )
{
  set u' :=  u ;
  0
  n Abs 1 -
  LOOP{
    set u' :=  u'RightShift ;
  };
  return u1' ;
}
```

```
: >>>          ABS 0 ?DO  >>  LOOP ;              ( u +n -- u' )
```

&<< **(u c -- u')**
LeftShiftWithCarry (u c -- u')

Logische Links-Verschiebung von *u* um eine Bit-Position mit Übertrag-Bit *c* . Dabei verfällt das MSB von *u* . Das Ergebnis wird als *u'* zurückgegeben. (Das Übertrag-Bit *c* wird dabei als Flag-Zustand ausgewertet, optimal sollten ihm aber die Zahlen 0 oder 1 übergeben werden.)

Schema:



Logische Links-Verschiebung
mit Übertrag C (Carry-Bit)

EVO PDF Tools Demo

```
def LeftShiftWithCarry( u , c ):( u' )
{
  set u' :=  Or( LeftShift( u ) , ToLsb( c )) ;
  return u' ;
}
```

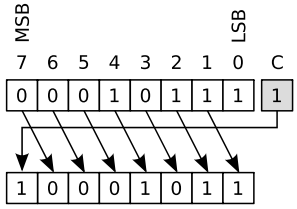
```
def LeftShiftWithCarry    ( u c -- u' )
{
  set u' :=  u LeftShift c ToLsb Or ;
  return u' ;
}
```

```
: &<<          >LSB SWAP << OR ;                  ( u c -- u' )
```

&>> **(u c -- u')**
RightShiftWithCarry (u c -- u')

Logische Rechts-Verschiebung von *u* um eine Bit-Position mit Übertrag-Bit *c* . Dabei verfällt das LSB von *u* . Das Ergebnis wird als *u'* zurückgegeben. (Das Übertrag-Bit *c* wird dabei als Flag-Zustand ausgewertet, optimal sollten ihm aber die Zahlen 0 oder 1 übergeben werden.)

Schema:



Logische Rechts-Verschiebung
mit Übertrag C (Carry-Bit)

```
def RightShiftWithCarry( u , c ):( u' )
{
  set u' :=  Or( RightShift( u ) , ToMsb( c )) ;
  return u' ;
}
```

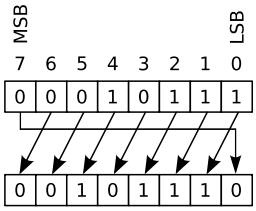
```
def RightShiftWithCarry    ( u c -- u' )
{
  set u' :=  u RightShift c ToMsb Or ;
  return u' ;
}
```

```
: &>>          >MSB SWAP >> OR ;                    ( u c -- u' )
```

C<< **(u -- u')**
CyclicLeftShift (u -- u')

Zyklische Links-Verschiebung der Bits von *u* um eine Bit-Position, wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben.

Schema:



Zyklische Links-Verschiebung

```
def CyclicLeftShift( u ):( u' )
{
  set u' :=  LeftShiftWithCarry( u , Lsb( u )) ;
  return u' ;
}
```

```
def CyclicLeftShift    ( u -- u' )
{
  set u' :=  u u Lsb LeftShiftWithCarry ;
  return u' ;
}
```

```
: C<<          DUP LSB C<<                          ( u -- u' )
```

EVO-PDF Tools Demo

C<<< **(u +n -- u')**
MultiCyclicLeftShift (u +n -- u')

Zyklische Links-Verschiebung der Bits von *u* um *+n* Bit-Positionen, wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben.

```
def MultiCyclicLeftShift( u , n ):( u' )
{
  set u' :=  u ;
  LOOP( 0 , Abs( n ) - 1 )
  {
    set u' :=  CyclicLeftShift( u' ) ;
  };
  return u' ;
}
```

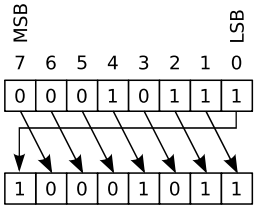
```
def MultiCyclicLeftShift    ( u n -- u' )
{
  set u' :=  u ;
  0
  n Abs 1 -
  LOOP{
    set u' :=  u' CyclicLeftShift ;
  };
  return u' ;
}
```

```
: C<<<         ABS 0 ?DO  C<<  LOOP ;               ( u +n -- u' )
```

C>> **(u -- u')**
CyclicRightShift (u -- u')

Zyklische Rechts-Verschiebung der Bits von *u* um eine Bit-Position, wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben.

Schema:



Zyklische Rechts-Verschiebung

```
def CyclicRightShift( u ):( u' )
{
  set u' :=  RightShiftWithCarry( u , Msb( u )) ;
  return u' ;
}
```

```
def CyclicRightShift    ( u -- u' )
{
  set u' :=  u u Msb RightShiftWithCarry ;
  return u' ;
}
```

```
: C>>          DUP MSB &>> ;                    ( u -- u' )
```

C>>> **(u +n -- u')**
MultiCyclicRightShift (u +n -- u')

Zyklische Rechts-Verschiebung der Bits von *u* um *+n* Bit-Positionen, wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben.

```
def MultiCyclicRightShift( u , n ):( u' )
{
  set u' :=  u ;
  LOOP( 0 , Abs( n ) - 1 )
  {
    set u' :=  CyclicRightShift( u' ) ;
  };
  return u' ;
}
```

```
def MultiCyclicRightShift    ( u n -- u' )
{
  set u' :=  u ;
  0
  n Abs 1 -
  LOOP{
    set u' :=  u' CyclicRightShift ;
  };
  return u' ;
}
```

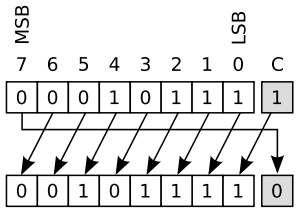
```
: C>>>         ABS 0 ?DO  C>>  LOOP ;            ( u +n -- u' )
```

EVO PDF Tools Demo

C&<< **(u c -- u' c')**
CyclicLeftShiftWithCarry (u c -- u' c')

Zyklische Links-Verschiebung der Bits von *u* um eine Bit-Position mit Übertrag-Bit *c* , wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben, und mit *c'* wird das MSB von *u* zurückgegeben.
(Das Übertrag-Bit *c* wird als Flag-Zustand ausgewertet, optimal sollten ihm aber die Zahlen 0 oder 1 übergeben werden.)

Schema:



Zyklische Links-Verschiebung
mit Übertrag C (Carry-Bit)

```
def CyclicLeftShiftWithCarry( u , c ):( u' , c' )
{
  set u' :=  LeftShiftWithCarry( u , c ) ;
  set c' :=  MsbToLsb( u ) ;
  return u' , c' ;
}
```

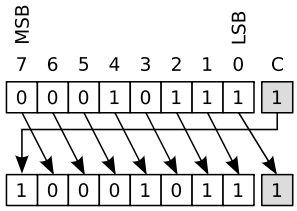
```
def CyclicLeftShiftWithCarry    ( u c -- u' c' )
{
  set u' :=  u c LeftShiftWithCarry ;
  set c' :=  u MsbToLsb ;
  return u' c' ;
}
```

```
: C&<<         OVER MSB>LSB >R &<< R> ;          ( u c -- u' c' )
```

C&>> **(u c -- u' c')**
CyclicRightShiftWithCarry (u c -- u' c')

Zyklische Rechts-Verschiebung der Bits von *u* um eine Bit-Position mit Übertrag-Bit *c* , wobei kein Bit verfällt. Das Ergebnis wird als *u'* zurückgegeben, und mit *c'* wird das LSB von *u* zurückgegeben.
(Das Übertrag-Bit *c* wird als Flag-Zustand ausgewertet, optimal sollten ihm aber die Zahlen 0 oder 1 übergeben werden.)

Schema:



Zyklische Rechts-Verschiebung
mit Übertrag C (Carry-Bit)

```
def CyclicRightShiftWithCarry( u , c ):( u' , c' )
{
  set u' :=  RightShiftWithCarry( u , c ) ;
  set c' :=  Lsb( u ) ;
  return u' , c' ;
}
```

```
def CyclicRightShiftWithCarry    ( u c -- u' c' )
{
  set u' :=  u c RightShiftWithCarry ;
  set c' :=  u Lsb ;
  return u' c' ;
}
```

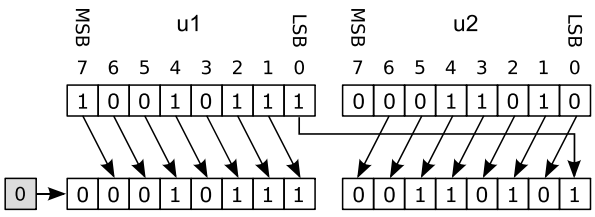
```
: C&>>          OVER LSB >R &>> R> ;            ( u c -- u' c' )
```

BEADSHIFT **(u1 u2 -- u1' u2')**
BeadShift (u1 u2 -- u1' u2')

EVO PDF Tools Demo

BEADSHIFT erwartet zwei Parameter (*u1* und *u2*). Dann wird *u1* logisch rechtsverschoben und das Resultat als Ergebniswert *u1'* zurückgegeben. Desweiteren wird *u2* logisch linksverschoben und das Resultat als Ergebnis *u2'* zurückgegeben; wobei zuvor noch das LSB von *u1* ins LSB von *u2'* übertragen wurde.
(Das MSB von *u2* verfällt.)

Schema:



Logische Rechts- / Links-Verschiebung
mit LSB-Übertragung (fürs Auffädeln)

```
def BeadShift( u1 , u2 ):( u1' , u2' )
{
  set u1' :=  RightShift( u1 ) ;
  set u2' :=  LeftShiftWithCarry( u2 , Lsb( u1 )) ;
  return u1' , u2' ;
}
```

```
def BeadShift    ( u1 u2 -- u1' u2' )
{
  set u1' :=  u1 RightShift ;
  set u2' :=  u2 u1 Lsb LeftShiftWithCarry ;
  return u1' u2' ;
}
```

```
: BEADSHIFT   OVER LSB &<< SWAP >> SWAP ;   ( u1 u2 -- u1' u2' )
```

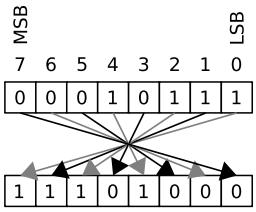
Das wiederholte Ausführen von BEADSHIFT ermöglicht eine umgekehrte Anordnung der Bits von *u1* im Rückgabewert *u2'* .

REVERSE **(u -- u')**

Reverse (u -- u')

Dreht die Reihenfolge der Bits in *u* um, d.h. vordere und hintere Bits werden gegeneinander vertauscht. Das Ergebnis wird als *u'* auf dem Stack gelegt.

Schema:



Reverse Bitreihenfolge

```
def Reverse( u ):( u' )
{
  set u' :=  0 ;
  LOOP( 0 , BitPerCell() - 1 )
  {
    set u , u' :=  BeadShift( u , u' ) ;
  };
  return u' ;
}
```

```
def Reverse    ( u -- u' )
{
  set u' :=  0 ;
  0
  BitPerCell 1 -
  LOOP{
    set u u' :=  u u' BeadShift ;
  };
  return u' ;
}
```

```
: REVERSE    0    BIT/CELL  0                          ( u -- u' )
             ?DO  BEADSHIFT   LOOP NIP ;
```

Bei beiden Pseudocodes ist die Zahl der Schleifendurchläufe die Anzahl der Bits pro Zelle. Dieser Wert ist jedoch um 1 zu mindern, weil der Schleifendurchlauf in bei beiden Pseudocodes mit dem Startwert 0 beginnt. Forth bietet uns diesbezüglich die bereits bekannte Erleichterung, weswegen wir bei Forth auf das Mindern um 1 auslassen können.

BIN **(--)**

(-- Nur für Forth ! --)

EVO PDF Tools Demo

Schaltet Forth in den binären Ein-/Ausgabe-Modus. Forth behandelt nun alle Zahlen als Binärzahlen.

```
: BIN          2 BASE ! ;                                ( -- )
```

UR.BIN **(u --)**

RightPrintBinU (u --)

Gibt *u* rechtsbündig als Binärzahl aus.

```
def RightPrintBinU( u ):()
{
  rightprint( u , BitPerCell() , base( 2 )) ;
}
```

```
def RightPrintBinU    ( u -- )
{
  u BitPerCell 2 base rightprint ;
}
```

```
: UR.BIN    BASE @ >R  BIN BIT/CELL U.R  R> BASE ! ;    ( u -- )
```

U.BIN **(u --)**

PrintBinU (u --)

Gibt *u* linksbündig als Binärzahl aus.

```
def PrintBinU( u ):()
{
  leftprint( u , base( 2 )) ;
}
```

```
def PrintBinU    ( u -- )
{
  u 2 base leftprint ;
}
```

```
: U.BIN        BASE @ >R  BIN U.  R> BASE ! ;          ( u -- )
```

U.OCT **(u --)**
PrintOctU (u --)

> Gibt *u* linksbündig als Oktalzahl aus.

```
def PrintOctU( u ):()
{
  leftprint( u , base( 8 )) ;
}
```

```
def PrintOctU    ( u -- )
{
  u 8 base leftprint ;
}
```

```
: U.OCT        BASE @ >R  OCTAL U.  R> BASE ! ;          ( u -- )
```

U.DEC **(u --)**
PrintDecU (u --)

> Gibt *u* linksbündig als Dezimalzahl aus.

```
def PrintDecU( u ):()
{
  leftprint( u , base( 10 )) ;
}
```

```
def PrintDecU    ( u -- )
{
  u 10 base leftprint ;
}
```

```
: U.DEC        BASE @ >R  DECIMAL U.  R> BASE ! ;        ( u -- )
```

EVO PDF Tools Demo

U.HEX **(u --)**
PrintHexU (u --)

> Gibt *u* linksbündig als Hexadezimalzahl aus.

```
def PrintHexU( u ):()
{
  leftprint( u , base( 16 )) ;
}
```

```
def PrintHexU    ( u -- )
{
  u 16 base leftprint ;
}
```

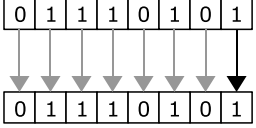
```
: U.HEX        BASE @ >R  HEX U.  R> BASE ! ;            ( u -- )
```

Studieren Sie jede einzelne der Routinen, bis Sie diese verstanden haben, und spielen Sie die internen Vorgänge jeder einzelnen Routine schematisch auf Papier durch, bis Sie die Wirkung der inneren Abläufe verstanden haben.

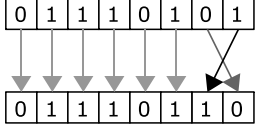
Wenn Sie die Wirkung jede dieser Routinen verstanden haben, sollten Sie in der Lage sein, eine Lösung für Übung 2) zu schreiben.

Übung 2)

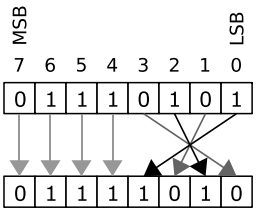
Definiere in Forth oder dem obigen Pseudocode eine sinnfreie antizyklische Verschiebung, der zwei Zahlen (*u* und *+n*) als Parameter übergeben werden. Bei Ausführung dieser Routine werden die *+n* niederwertigsten Bit der Zahl *u* antizyklisch verschoben. Die Wirkungsweise dieser antizyklischen Bit-Verschiebung ist in der nachfolgenden Abbildung skizziert und in Ihrem Codeentwurf passend zu formulieren.



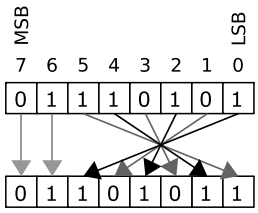
Antizyklische Verschiebung
z.B. um 1 Bit



Antizyklische Verschiebung
z.B. um 2 Bit



Antizyklische Verschiebung
z.B. um 4 Bit



Antizyklische Verschiebung
z.B. um 6 Bit

Die obigen Bit-Operatotionen ausprobieren

Zum Selbststudium und Ausprobieren der obigen Routinen haben wir den dazu notwendigen Programmcode auf Grundlage der Programmiersprache Forth (Forth83) für Sie niedergeschrieben.

Starten Sie Ihr Forth-System (Forth83), tippen Sie den Inhalt des nachstehenden Screen-Listings ab und speichern Sie die abgetippten Screen-Inhalte auf Ihren Datenträger ab. Bei dem hier dargestellten Screen-Listing gehen wir davon aus, daß dieses in den Blöcken von Blocknummer 210 bis Blocknummer 249 (inklusive) abgespeichert ist.

Mit der Anweisung

210 LOAD (anschließend bitte die Enter-Taste drücken)

EVO PDF Tools Demo

liest Forth den in Block 210 hinterlegten Code ein, kompiliert diesen und führt ihn aus. In Zeile 16 (von Block 210) trifft Forth dabei auf die Anweisung `-->` . Diese weist Forth an, den Code des numerisch nächsten Blocks einzulesen, zu kompilieren auszuführen. Jedes mal wenn Forth dabei auf die Anweisung `-->` trifft, liest es den jeweils nächsten numerischen Block ein, kompiliert diesen und führt dessen Code aus. Derart arbeitet Forth mittels `-->` sämtliche der nachstehenden Codeblöcke bis Blocknummer 249 (inklusive) ab. Anschließend sind die in diesen Blöcken definierten Routinen in den Arbeitsspeicher ihres Computers kompiliert und liegen dort ausführbar vor. Nun können Sie jede dieser Route interaktiv ausprobieren und in den Programmcode eigener Forth-Routinen einfließen lassen. Testen Sie das ganze nach Herzenslust aus.

```
SCR # 210
  1   ( +-------------------------------------------------------+ )
  2   ( |                BIT- UND FLAG-OPERATIONEN                | )
  3   ( +-------------------------------------------------------+ )
  4   (                                                           )
  5   ( Wir stellen Ihnen hier  eine Lehrsammlung von Bit- und Flag- )
  6   ( Operatoren vor,  um Sie mit diesem Themenbereich vertraut zu )
  7   ( machen.  Dabei fuehren wir Sie in  die Philosophie von Forth )
  8   ( ein, wie es Programmieraufgaben strukturiert und loest.      )
  9   (                                                           )
 10   ( Einige Woerter, die wir Ihnen in dieser Sammlung vorstellen, )
 11   ( werden in Ihrem Forth-System bereits enthalten sein. Dennoch )
 12   ( legen wir manche von diesen neu an, um Ihnen ein lueckenlos- )
 13   ( ses Veraendnis dieses  Themenbereichs zu ermoeglichen.  Dazu )
 14   ( haben  wir Ihnen  die Woerter  so definiert,  dass Sie diese )
 15   ( einzeln ausprobieren und leicht verstehen koennen.           )
 16   -->
```

```
SCR # 211
  1   ( Forth vorbereiten )
  2
  3   WARNING OFF                ( Kompiler-Hinweise abschalten   )
  4   DECIMAL                    ( dezimalen E/A-Modus aktivieren )
  5
  6
  7
  8
  9
 10
```

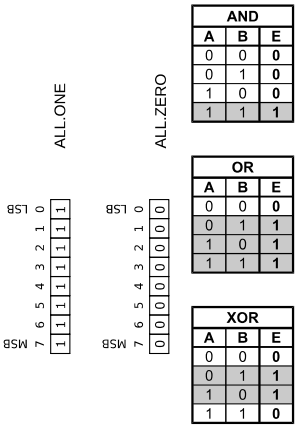
```
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 212
  1   ( 1.  Hardware Abhaengigkeiten                                  )
  2   (                                                               )
  3   (   BIT/BYTE     -- n       Liefert Anzahl der Bits je Byte.    )
  4   (                                                               )
  5   (   BYTE/CELL    -- n       Liefert Anzahl der Bytes je Zelle.  )
  6   (                                                               )
  7   (                                                               )
  8   (   Den Speicherraum, den Forth zur Aufnahme einer Integer      )
  9   (   benoetigt, wird in Forth als "Zelle" bezeichnet.            )
 10   (                                                               )
 11   (                                                               )
 12   (                                                               )
 13   (                                                               )
 14   (                                                               )
 15   (                                                               )
 16   -->
```

```
SCR # 213
  1   ( 1.  Hardware Abhaengigkeiten )
  2
  3
  4   8            ( Anzahl der Bit je Byte, Wert ggf. anpassen! )
  5   CONSTANT BIT/BYTE                              ( -- Bit_je_Byte )
  6
  7
  8     CREATE =MARKE=                 (   Hilfskonstrukt =MARKE= )
  9       HERE 0 , HERE - ABS          ( Ermittelt Bytes je Zelle )
 10     FORGET =MARKE=                 (   =MARKE= nun vergessen! )
 11   ( Auf dem Stack liegt nun die )
 12   ( Anzahl der Bytes je Zelle.  )
 13   CONSTANT BYTE/CELL                            ( -- Bytes_je_Zelle )
 14
 15
 16   -->
```

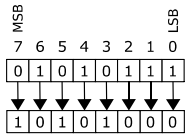
EVO PDF Tools Demo

```
SCR # 214
  1   ( 2.  Bit- und Flag-Operationen                                 )
  2   (                                                               )
  3   ( 2.1.  Grundlegende Bitmasken                                  )
  4   (                                                               )
  5   (   ALL.ONE      -- u       Bei der Zahl u sind alle Bits 1.    )
  6   (   ALL.ZERO     -- u       Bei der Zahl u sind alle Bits 0.    )
  7   (                                                               )
  8   ( 2.2.  Grundlegende bitweise Operatoren                        )
  9   (                                                               )
 10   (   AND      u1 u2 -- u3      Liefert u3 als Ergebnis der AND- )
 11   (                               Verknuepfung von u1 und u2.    )
 12   (   OR       u1 u2 -- u3      Liefert u3 als Ergebnis der OR-  )
 13   (                               Verknuepfung von u1 und u2.    )
 14   (   XOR      u1 u2 -- u3      Liefert u3 als Ergebnis der XOR- )
 15   (                               Verknuepfung von u1 und u2.    )
 16   -->
```



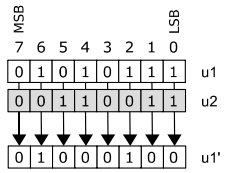

```
SCR # 215
  1   ( 2.  Bit- und Flag-Operationen )
  2
  3   ( 2.1.  Grundlegende Bitmasken )
  4
  5   -1  CONSTANT ALL.ONE                        ( -- alle_Bit_sind_1 )
  6    0  CONSTANT ALL.ZERO                       ( -- alle_Bis_sind_0 )
  7
  8
  9   ( 2.2.  Grundlegende bitweise Operatoren )
 10
 11   ( AND       //vordefiniert////                    { u1 u2 -- u3 }
 12   ( OR        //vordefiniert////                    { u1 u2 -- u3 }
 13   ( XOR       //vordefiniert////                    { u1 u2 -- u3 }
 14
 15
 16   -->
```

```
SCR # 216
  1   ( 2.3.  Bitweise Operationen mit Invertierung                   )
  2   (                                                               )
  3   ( 2.3.1.  Einer-Komplement                                      )
  4   (                                                               )
  5   (   INVERT    u -- u'          Invertiert alle Bits von u; d.h. )
```



```
  6   (                                jede 0 wird zu 1 veraendert,    )
  7   (                                und jede 1 wird zu einer 0.     )
  8   (                                                                )
  9   ( 2.3.2.  Inhibition                                             )
 10   (                                                                )
 11   (   INHIBIT   u1 u2 -- u1'    Das Ergebnis u3 uebernimmt alle    )
 12   (                                Bits von u1; wobei aber jene    )
 13   (                                Bit-Positionen genullt werden,  )
 14   (                                die in u2 den Wert 1 haben.     )
 15   (                                                                )
 16   -->
```

Invertieren aller Bits



Inhibitieren ausgewählter Bits

```
SCR # 217
  1   ( 2.3.  Bitweise Operationen mit Invertierung )
  2
  3   ( 2.3.1  Einer-Komplement )
  4
  5   : INVERT       ALL.ONE XOR ;                        ( u -- u' )
  6
  7
  8   ( 2.3.2  Inhibition )
  9
 10   : INHIBIT      INVERT AND ;                     ( u1 u2 -- u1' )
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 218
  1   ( 2.3.3.  Operationen mit invertiertem Ergebnis                  )
  2   (                                                                )
  3   (   NAND     u1 u2 -- u3      Invertierte AND-Operation von     )
  4   (                               u1 und u2; liefert u3.           )
  5   (                                                                )
  6   (   NOR      u1 u2 -- u3      Invertierte OR-Operation von      )
  7   (                               u1 und u2; liefert u3.           )
  8   (                                                                )
  9   (   XNOR     u1 u2 -- u3      Invertierte XOR-Operation von     )
 10   (                               u1 und u2; liefert u3.           )
 11   (                                                                )
 12   (                                                                )
 13   (                                                                )
 14   (                                                                )
 15   (                                                                )
 16   -->
```

EVO PDF Tools Demo

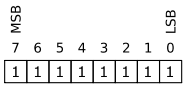
NAND		
A	B	E
0	0	1
0	1	1
1	0	1
1	1	0

NOR		
A	B	E
0	0	1
0	1	0
1	0	0
1	1	0

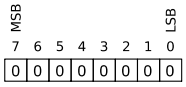
XNOR		
A	B	E
0	0	1
0	1	0
1	0	0
1	1	1

```
SCR # 219
  1   ( 2.3.3.  Operationen mit invertiertem Ergebnis )
  2
  3   : NAND         AND INVERT ;                     ( u1 u2 -- u3 )
  4   : NOR          OR  INVERT ;                     ( u1 u2 -- u3 )
  5   : XNOR         XOR INVERT ;                     ( u1 u2 -- u3 )
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 220
  1   ( 2.4.  Boolesche Wahrheitswerte fuer Flags                      )
  2   (                                                                )
  3   (   TRUE     { -- TRUE }      Liefert die Bitmaske fuer TRUE.    )
  4   (   FALSE    { -- FALSE }     Liefert die Bitmaske fuer FALSE.   )
  5   (                                                                )
  6   ( 2.4.1.  Wahrheitswerte umkehren                                )
  7   (   NOT      u -- f      Liest u als Flag und liefert dessen     )
  8   (                          negierten Wahrheitswert als echtes    )
  9   (                          Boolesches Flag {TRUE oder FALSE}.    )
 10   (                                                                )
 11   ( 2.4.3.  Wahrheitswerte konsolidieren                           )
 12   (   BOOL    u -- f       Liest u als Flag und liefert dessen     )
 13   (                          konsolidierten  Wahrheitswert als     )
 14   (                          Boolesches Flag {TRUE oder FALSE}.    )
 15   (                                                                )
 16   -->
```



TRUE



FALSE

```
SCR # 221
```

```
  1   ( 2.4.  Boolesche Wahrheitswerte fuer Flags )
  2
  3   ALL.ONE   CONSTANT TRUE                              ( -- TRUE )
  4   ALL.ZERO  CONSTANT FALSE                            ( -- FALSE )
  5
  6   ( 2.4.1.  Wahrheitswerte umkehren )
  7
  8   : NOT          FALSE = ;                              ( u -- f )
  9            ( Wort = ist vordefiniert )
 10
 11   ( 2.4.3.  Wahrheitswerte konsolidieren )
 12
 13   : BOOL         NOT NOT ;                              ( u -- f )
 14            ( Trick: doppelte Verneinung )
 15
 16   -->
```

```
SCR # 222
  1   ( 2.5.  Vorzeichenbehaftete Zahlen vergleichen                 )
  2   (                                                              )
  3   (   <      n1 n2 -- f    Ist TRUE, wenn n1 kleiner als n2 ist. )
  4   (   >      n1 n2 -- f    Ist TRUE, wenn n1 groesser als n2 ist.)
  5   (   =      n1 n2 -- f    Ist TRUE, wenn n1 gleich n2 ist.      )
  6   (   <>     n1 n2 -- f    Ist TRUE, wenn n1 ungleich n2 ist.    )
  7   (                                                              )
  8   (                                                              )
  9   ( 2.6.  Vorzeichenlose Zahlen vergleichen                      )
 10   (                                                              )
 11   (   U<     u1 u2 -- f    Ist TRUE, wenn u1 kleiner als u2 ist. )
 12   (   U>     u1 u2 -- f    Ist TRUE, wenn u1 groesser als u2 ist.)
 13   (   U=     u1 u2 -- f    Ist TRUE, wenn u1 gleich u2 ist.      )
 14   (   U<>    u1 u2 -- f    Ist TRUE, wenn u1 ungleich u2 ist.    )
 15   (                                                              )
 16   -->
```

```
SCR # 223
  1   ( 2.5.  Vorzeichenbehaftete Zahlen vergleichen )
  2
  3   ( <            //vordefiniert////                 { n1 n2 -- f )
  4   : >            SWAP < ;                            ( n1 n2 -- f )
  5   ( =            //vordefiniert////                 ( n1 n2 -- f )
  6   : <>           = NOT ;                             ( n1 n2 -- f )
  7
  8
  9   ( 2.6.  Vorzeichenlose Zahlen vergleichen )
 10
 11   ( U<           //vordefiniert////                 { u1 u2 -- f )
 12   : U>           SWAP U< ;                           ( u1 u2 -- f )
 13   : U=           = ;                                 ( u1 u2 -- f )
 14   : U<>          <> ;                                ( u1 u2 -- f )
 15
 16   -->
```

EVO PDF Tools Demo

```
SCR # 224
  1   ( 2.7.  Vorzeichenbehaftete Zahlen gegen 0 vergleichen         )
  2   (   0=      n -- f        Ist TRUE, wenn n gleich 0 ist.       )
  3   (   0<>     n -- f        Ist TRUE, wenn n ungleich 0 ist.     )
  4   (   0<      n -- f        Ist TRUE, wenn n kleiner als 0 ist.  )
  5   (   0>      n -- f        Ist TRUE, wenn n groesser als 0 ist. )
  6   (                                                              )
  7   ( 2.8.  Vorzeichen-Operationen fuer Integer                    )
  8   (   POSITIVE?    n -- f      Ist TRUE, wenn n1 eine positive   )
  9   (                              Zahl ist.                       )
 10   (   NEGATIVE?    n -- f      Ist TRUE, wenn n1 eine negative   )
 11   (                              Zahl ist.                       )
 12   (   NEGATE       n -- n'     Aendert das Vorzeichen von n. Das )
 13   (                              entspricht einer Multiplikation )
 14   (                              von n mit -1.                   )
 15   (                                                              )
 16   -->
```

```
SCR # 225
  1   ( 2.7.  Vorzeichenbehaftete Zahlen gegen 0 vergleichen )
  2
  3   : 0=           0 = ;                                  ( n -- f )
  4   : 0<>          0 <> ;                                 ( n -- f )
  5   : 0>           0 > ;                                  ( n -- f )
  6   : 0<           0 < ;                                  ( n -- f )
  7
  8
  9   ( 2.8.  Vorzeiche-Operationen bei Integer )
 10
 11   : POSITIVE?    0> ;                                   ( n -- f )
 12   : NEGATIVE?    0< ;                                   ( n -- f )
```

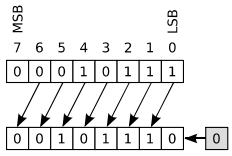
```
 13
 14   : NEGATE       INVERT 1+ ;                          ( n -- n' )
 15
 16   -->
```

```
SCR # 226
  1   ( 2.9.  Umrechnungen zwischen Bit und Byte                    )
  2   (                                                             )
  3   (   BYTE>BIT     +n1 -- n2      Rechnet +n1 Bytes in Bits um.  )
  4   (   BIT>BYTE     +n1 -- n2      Rechnet +n1 Bits in Bytes um.  )
  5   (                                                             )
  6   (   Bei den Umrechnungen von Bit und Byte wird von einer       )
  7   (   Hardwarekonfiguration von 8 Bit pro Byte ausgegangen.      )
  8   (   Andernfalls ist dieses im Code zu aendern!                 )
  9   (                                                             )
 10   (                                                             )
 11   (   BIT/CELL     -- n       Liefert Anzahl der Bits  pro Zelle )
 12   (                                                             )
 13   (                                                             )
 14   (                                                             )
 15   (                                                             )
 16   -->
```

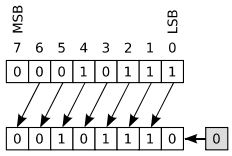
```
SCR # 227
  1   ( 2.9.  Umrechnungen zwischen Bit und Byte )
  2
  3   : BYTE>BIT     ABS BIT/BYTE * ;          ( Bytezahl -- Bitzahl )
  4
  5     ( * Trick: Aufrundung der Bytezahl via Floor-Division  * )
  6     ( *        bei negativen Devisor,  und Ruecknegierung  * )
  7     ( *        des Divisionsergebnisses.                   * )
  8   : BIT>BYTE                                ( Bitzahl -- Bytezahl )
  9                      ABS       ( positiver Divident )
 10          BIT/BYTE NEGATE       ( negierter Devisor  )
 11                        /       ( Floor-Division     )
 12                   NEGATE  ;    ( ruecknegiertes Ergebnis )
 13
 14   BYTE/CELL BYTE>BIT   CONSTANT BIT/CELL       ( -- Bit_je_Zelle )
 15
 16   -->
```

```
SCR # 228
  1   ( 2.10.  Logische Links-Verschiebung /                        )
  2   (        Logical Left Shift                                   )
  3   (                                                             )
  4   (   <<       u -- u'        Logische Links-Verschiebung um     )
  5   (                             1 Bit. Dabei verfaellt das MSB  )
  6   (                             von u.                          )
  7   (                                                             )
  8   (   <<<      u +n -- u'     Logische Links-Verschiebung um     )
  9   (                             +n Bits. Dabei verfallen u.U.   )
 10   (                             mehrere MSBs von u.             )
 11   (                                                             )
 12   (    Die logische und die arithmetische Links-Verschiebung    )
 13   (    sind ein und dieselbe Bit-Operation.  Somit deckt die    )
 14   (    Definition der logischen Links-Verschiebung beide ab.    )
 15   (                                                             )
 16   -->
```

EVO PDF Tools Demo



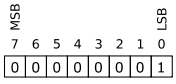
Arithmetische Links-Verschiebung



Logische Links-Verschiebung

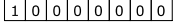
```
SCR # 229
  1   ( 2.10.  Logische Links-Verschiebung )
  2
  3   : <<       2* ;                                   ( u -- u' )
  4   : <<<      ABS 0 ?DO  <<  LOOP ;                 ( u +n -- u' )
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 230
  1   ( 2.11.  Ganzzahl-Bitmasken                                   )
  2   (                                                             )
  3   (   LSB.ONE     -- u         Bitmaske zur Isolierung des LSB. )
  4   (                               Bei den Bits dieser Zahl ist  )
  5   (                               nur das LSB 1; die uebrigen   )
  6   (                               Bit sind 0.                   )
  7   (                                                             )
  8   (   MSB.ONE     -- u         Bitmaske zur Isolierung des MSB. )
```

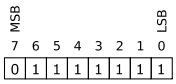


Nur LSB ist 1

```
  8   (   MSB.ONE     -- u          Bitmaske zur Isolierung des MSB.    )
  9   (                                Bei den Bits dieser Zahl ist     )
 10   (                                nur das MSB 1; die uebrigen      )
 11   (                                Bit sind 0.                      )
 12   (                                                                 )
 13   (   INT.MAX     -- +n         Legt die groesste positive Integer )
 14   (                                des Forth-Systems auf dem Stack )
 15   (                                ab.                              )
 16   -->
```



Nur MSB ist 1

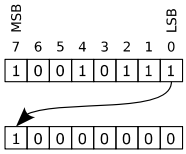


größter positiver Integerwert

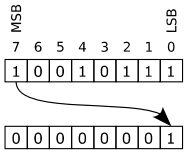
```
SCR # 231
  1   ( 2.11.  Ganzzahl-Bitmasken )
  2
  3   1 CONSTANT LSB.ONE                          ( -- nur_LSB_ist_1 )
  4
  5   LSB.ONE   BIT/CELL 1-  <<<
  6   CONSTANT MSB.ONE                            ( -- nur_MSB_ist_1 )
  7
  8   MSB.ONE INVERT
  9   CONSTANT INT.MAX                ( -- alle_Bit_ausser_MSB_sind_1 )
 10
 11     ( Alternativ zur Anweisung     MSB.ONE INVERT              )
 12     ( kann  auch die Anweisung     MSB.ONE 1-                  )
 13     ( genutzt werden, da beide das gleiche Bitmuster liefern. )
 14
 15
 16   -->
```

```
SCR # 232
  1   ( 2.12.  MSB- und LSB-Operationen                              )
  2   (   LSB        u -- u'      Liefert nur das LSB der Zahl u;    )
  3   (                             die uebrigen Bit von u' sind 0.  )
  4   (   MSB        u -- u'      Liefert nur das MSB der Zahl u;    )
  5   (                             die uebrigen Bit von u' sind 0.  )
  6   (   >LSB       f -- u       Legt Wahrheitswert des Flag f im   )
  7   (                             LSB von u ab;  dessen  uebrige   )
  8   (                             Bit alle auf 0 gesetzt sind.     )
  9   (   >MSB       f -- u       Legt Wahrheitswert des Flag f im   )
 10   (                             MSB von u ab;  dessen  uebrige   )
 11   (                             Bit alle auf 0 gesetzt sind.     )
 12   (   LSB>MSB    u -- u'      Das LSB von u wird zu MSB von u';  )
 13   (                             die uebrigen Bit von u' sind 0.  )
 14   (   MSB>LSB    u -- u'      MSB von u wird zu LSB, uebr. Bit 0.)
 15   (                             die uebrigen Bit von u' sind 0.  )
 16   -->
```

EVO PDF Tools Demo



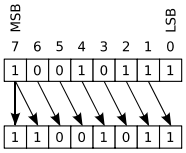
LSB>MSB



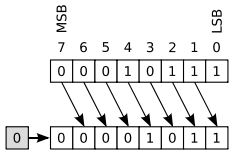
MSB>LSB

```
SCR # 233
  1   ( 2.12.  MSB- und LSB-Operationen )
  2
  3   : MSB          MSB.ONE AND ;                        ( u -- u' )
  4   : LSB          LSB.ONE AND ;                        ( u -- u' )
  5
  6   : >MSB         BOOL MSB ;                            ( f -- u )
  7   : >LSB         BOOL LSB ;                            ( f -- u )
  8
  9   : LSB>MSB      LSB >MSB ;                           ( u -- u' )
 10   : MSB>LSB      MSB >LSB ;                           ( u -- u' )
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 234
  1   ( 2.13.  Arithmetisches Rechts-Verschiebung /                  )
  2   (        Arthimetic Right-Shift                                )
  3   (                                                              )
  4   (   A>>      n -- n'        Arithmetische Rechts-Verschiebung  )
  5   (                             um 1 Bit. Das LSB verfaellt.     )
  6   (   A>>>     n1 +n2 -- n1'  Arithmetische Rechts-Verschiebung  )
  7   (                             um +n2 Bits; u.U. verfallen LSBs.)
  8   (                                                              )
  9   ( 2.14.  Logische Rechts-Verschiebung /                        )
 10   (        Logical Right-Shift                                   )
 11   (                                                              )
 12   (   >>       u -- u'        Logische Rechts-Verschiebung um    )
 13   (                             1 Bit. Das LSB verfaellt.        )
 14   (   >>>      u +n -- u'     Logische Rechts-Verschiebung um    )
 15   (                             +n Bits; u.U. verfallen LSBs.    )
 16   -->
```



Arithmetische Rechts-Verschiebung

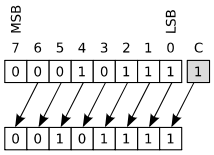


Logische Rechts-Verschiebung

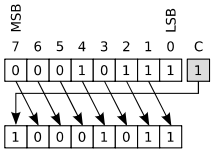
```
SCR # 235
  1   ( 2.13.  Arithmetisches Rechts-Verschiebung )
  2
```

```
  3   : A>>          2/ ;                                  ( n -- n' )
  4   : A>>>         ABS 0 ?DO  A>>  LOOP ;          ( n1 +n2 -- n1' )
  5
  6
  7   ( 2.14.  Logische Rechts-Verschiebung )
  8
  9   : >>           2/ INT.MAX AND ;                      ( u -- u' )
 10   : >>>          ABS 0 ?DO  >>  LOOP  ;             ( u +n -- u' )
 11
 12   ( Bei der arithmeischen Rechts-Verschiebung bleibt das Vorzei- )
 13   ( chen-Bit [MSB] erhalten, bei der logischen hingegen nicht.   )
 14
 15
 16   -->
```

```
SCR # 236
  1   ( 2.15.  Logische Links-Verschiebung mit Uebertrag-Bit /       )
  2   (        Logical Left Shift with Carry                         )
  3   (                                                              )
  4   (   &<<      u c -- u'      Logische Links-Verschiebung um     )
  5   (                             1 Bit mit Uebertrag c. Dabei     )
  6   (                             verfaellt das MSB von u.         )
  7   (                                                              )
  8   (                                                              )
  9   ( 2.16.  Logische Rechts-Verschiebung mit Uebertrag-Bit /      )
 10   (        Logical Right Shift with Carry                        )
 11   (                                                              )
 12   (   &>>      u c -- u'      Logische Rechts-Verschiebung um    )
 13   (                             1 Bit mit Uebertrag c. Dabei     )
 14   (                             verfaellt das LSB von u.         )
 15   (                                                              )
 16   -->
```



Logische Links-Verschiebung mit Übertrag C (Carry-Bit)

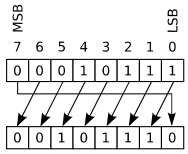


Logische Rechts-Verschiebung mit Übertrag C (Carry-Bit)

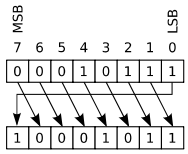
```
SCR # 237
  1   ( 2.15.  Logische Links-Verschiebung mit Uebertrag-Bit )
  2
  3   : &<<          >LSB SWAP << OR ;                   ( u c -- u' )
  4                                          ( c steht für Carry-Bit )
  5
  6
  7
  8   ( 2.16.  Logische Rechts-Verschiebung mit Uebertrag-Bit )
  9
 10   : &>>          >MSB SWAP >> OR ;                   ( u c -- u' )
 11                                          ( c steht für Carry-Bit )
 12
 13
 14
 15
 16   -->
```

EVO PDF Tools Demo

```
SCR # 238
  1   ( 2.17.  Zyklische Links-Verschiebung /                        )
  2   (        Cyclic Left Shift                                     )
  3   (                                                              )
  4   (   C<<      u -- u'        Zyklische Links-Verschiebung um    )
  5   (                             1 Bit. Kein Bit verfaellt.       )
  6   (   C<<<     u +n -- u'     Zyklische Links-Verschiebung um    )
  7   (                             +n Bits. Kein Bit verfaellt.     )
  8   (                                                              )
  9   ( 2.18.  Zyklische Rechts-Verschiebung /                       )
 10   (        Cyclic Right Shift                                    )
 11   (                                                              )
 12   (   C>>      u -- u'        Zyklische Rechts-Verschiebung um   )
 13   (                             1 Bit. Kein Bit verfaellt.       )
 14   (   C>>>     u +n -- u'     Zyklische Rechts-Verschiebung um   )
 15   (                             +n Bits. Kein Bit verfaellt.     )
 16   -->
```



Zyklische Links-Verschiebung

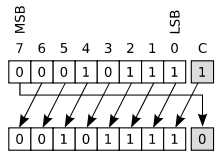


Zyklische Rechts-Verschiebung

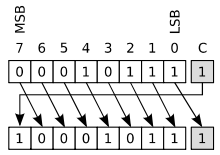
```
SCR # 239
  1   ( 2.17.  Zyklische Links-Verschiebung )
  2
  3   : C<<          DUP LSB &<< ;                         ( u -- u' )
  4   : C<<<         ABS 0 ?DO  C<<  LOOP ;              ( u +n -- u' )
  5
  6
  7
  8   ( 2.18.  Zyklische Rechts-Verschiebung )
  9
 10   : C>>          DUP MSB &>> ;                         ( u -- u' )
 11   : C>>>         ABS 0 ?DO  C>>  LOOP ;              ( u +n -- u' )
 12
 13
 14
 15
```

```
 16   -->
```

```
SCR # 240
  1   ( 2.19.  Zyklische Links-Verschiebung mit Uebertrag-Bit /      )
  2   (        Cyclic Left Shift with Carry Bit                      )
  3   (                                                              )
  4   (   C&<<     u c -- u' c'     Zyklische Links-Verschiebung um  )
  5   (                              1 Bit mit Uebertrag-Bit c.      )
  6   (                              Kein Bit verfaellt.             )
  7   (                                                              )
  8   (                                                              )
  9   ( 2.20.  Zyklische Rechts-Verschiebung mit Uebertrag-Bit /     )
 10   (        Cyclic Right Shift with Carry Bit                     )
 11   (                                                              )
 12   (   C&>>     u c -- u' c'     Zyklische Rechts-Verschiebung um )
 13   (                              1 Bit mit Uebertrag-Bit c.      )
 14   (                              Kein Bit verfaellt.             )
 15   (                                                              )
 16   -->
```



Zyklische Links-Verschiebung mit Übertrag C (Carry-Bit)

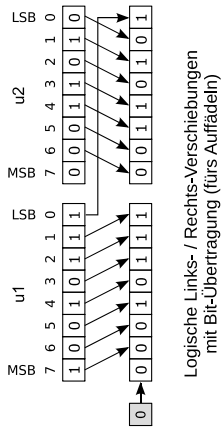


Zyklische Rechts-Verschiebung mit Übertrag C (Carry-Bit)

```
SCR # 241
  1   ( 2.19.  Zyklische Links-Verschiebung mit Uebertrag-Bit )
  2
  3   : C&<<         OVER MSB>LSB >R  &<< R> ;        ( u c -- u' c' )
  4
  5
  6
  7   ( 2.20.  Zyklische Rechts-Verschiebung mit Uebertrag-Bit )
  8
  9   : C&>>         OVER LSB >R  &>> R> ;            ( u c -- u' c' )
 10
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 242
  1   ( 2.21.  Bitweise "Perlenketten"-Aufreihung                    )
  2   (                                                              )
  3   (   BEADSHIFT    u1 u2 -- u1' u2'                              )
  4   (                                                              )
  5   (   BEADSHIFT erwartet zwei Zahlen (u1 und u2) auf dem Stack.  )
  6   (   Dann wird u1 logisch rechtsverschoben und als u1' zurueck- )
  7   (   gegeben. Desweiteren wird u2 linksverschoben und als u2'   )
  8   (   zurueckgegeben, wobei zuvor noch das LSB von u1 in das LSB )
  9   (   von u2' uebertragen wurde. Das MSB von u2 verfaellt.       )
 10   (                                                              )
 11   (   Hinweis:  Das wiederholte Ausfuehren von BEADSHIFT ermoeg- )
 12   (             licht eine umgekehrte Anordnung der Bits von u1  )
 13   (             im Rueckgabewert u2'.                            )
 14   (                                                              )
 15   (                                                              )
 16   -->
```

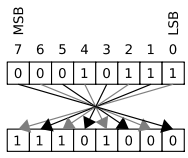
EVO PDF Tools Demo



Logische Links- / Rechts-Verschiebungen mit Bit-Übertragung (fürs Auffädeln)

```
SCR # 243
  1   ( 2.21.  Bitweise "Perlenketten"-Aufreihung )
  2
  3   : BEADSHIFT                                  ( u1 u2 -- u1' u2' )
  4         ( u1  u2          )    OVER
  5         ( u1  u2  u1      )     LSB
  6         ( u1  u2  u1_lsb  )     &<<
  7         ( u1  u2'         )    SWAP
  8         ( u2' u1          )      >>
  9         ( u2' u1'         )    SWAP
 10   ;     ( u1' u2'         )
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 244
  1   ( 2.22.  Bitreihenfolge umkehren                               )
  2   (                                                              )
  3   (   REVERSE  u -- u'     Dreht die Reihenfolge der Bits in u   )
  4   (                          spiegelverkehrt um,  d.h. vordere   )
  5   (                          und hintere Bits werden gegenein-   )
  6   (                          ander vertauscht.   Das  Ergebnis   )
  7   (                          ist u'.                             )
  8   (                                                              )
  9   (                                                              )
 10   (                                                              )
```



Reverse Bitreihenfolge

```
 11   (                                                                )
 12   (                                                                )
 13   (                                                                )
 14   (                                                                )
 15   (                                                                )
 16   -->
```

```
SCR # 245
  1   ( 2.22.  Bitreihenfolge umkehren )
  2
  3   : REVERSE                                              ( u -- u' )
  4       ( u                )    0
  5       ( u   0            )    BIT/CELL  0
  6       ( u   0 ende start )    ?DO
  7       ( u   0            )           BEADSHIFT
  8       ( u1' u'           )    LOOP
  9       ( u1' u'           )    NIP
 10   ;    ( u'               )
 11
 12
 13
 14
 15
 16   -->
```

```
SCR # 246
  1   ( 3.  Ein-/Ausgasbe-Woerter                                      )
  2   (                                                                )
  3   (   BIN         --          Aktiviert den Ein-/Ausgabe-Modus     )
  4   (                             fuer Binaerzahlen [Dualzahlen].    )
  5   (   UR.BIN      u --        Gibt u rechtsbuendig als Binaer-     )
  6   (                             zahl aus.                          )
  7   (   U.BIN       u --        Gibt u linksbuendig als Binaer-      )
  8   (                             zahl aus.                          )
  9   (   U.OCTAL     u --        Gibt u linksbuendig als Oktal-       )
 10   (                             zahl aus.                          )
 11   (   U.DECIMAL   u --        Gibt u linksbuendig als Dezimal-     )
 12   (                             zahl aus.                          )
 13   (   U.HEX       u --        Gibt u linksbuendig als Hexa-        )
 14   (                             dezimalzahl aus.                   )
 15   (                                                                )
 16   -->
```

EVO PDF Tools Demo

```
SCR # 247
  1   ( 3.  Ein-/Ausgasbe-Woerter )
  2
  3   : BIN          2 BASE ! ;                                ( -- )
  4
  5   : UR.BIN       BASE @ >R                                ( u -- )
  6                  BIN BIT/CELL U.R
  7                  R> BASE ! ;
  8
  9   : U.BIN        BASE @ >R  BIN     U.  R> BASE ! ;       ( u -- )
 10   : U.OCT        BASE @ >R  OCTAL   U.  R> BASE ! ;       ( u -- )
 11   : U.DEC        BASE @ >R  DECIMAL U.  R> BASE ! ;       ( u -- )
 12   : U.HEX        BASE @ >R  HEX     U.  R> BASE ! ;       ( u -- )
 13
 14
 15
 16   -->
```

```
SCR # 248
  1   ( Hiermit ist die Lehrsammlung der Bit- und Flag- Woerter        )
  2   ( abgeschlossen.                                                 )
  3   (                                                                )
  4   ( Mit dem naechsten Block rauemen wir Ihr Forth-System auf,      )
  5   ( so dass Sie anschliessend die Woerter dieser Sammlung nach     )
  6   ( Belieben austesten konnen.                                     )
  7   (                                                                )
  8   ( Dabei wuenschen wir Ihnen viel Spass!                          )
  9   (                                                                )
 10   (                                                                )
 11   (                                                                )
 12   (                                                                )
 13   (                                                                )
 14   (                                                                )
 15   (                                                                )
 16   -->
```

```
SCR # 249
  1   ( Forth aufraeumen )
  2
  3   WARNING ON                ( Kompiler-Hinweise aktivieren   )
  4   DECIMAL                   ( dezimalen E/A-Modus aktivieren )
  5
```

6
7
8
9
10
11
12
13
14
15
16

EVO PDF Tools Demo